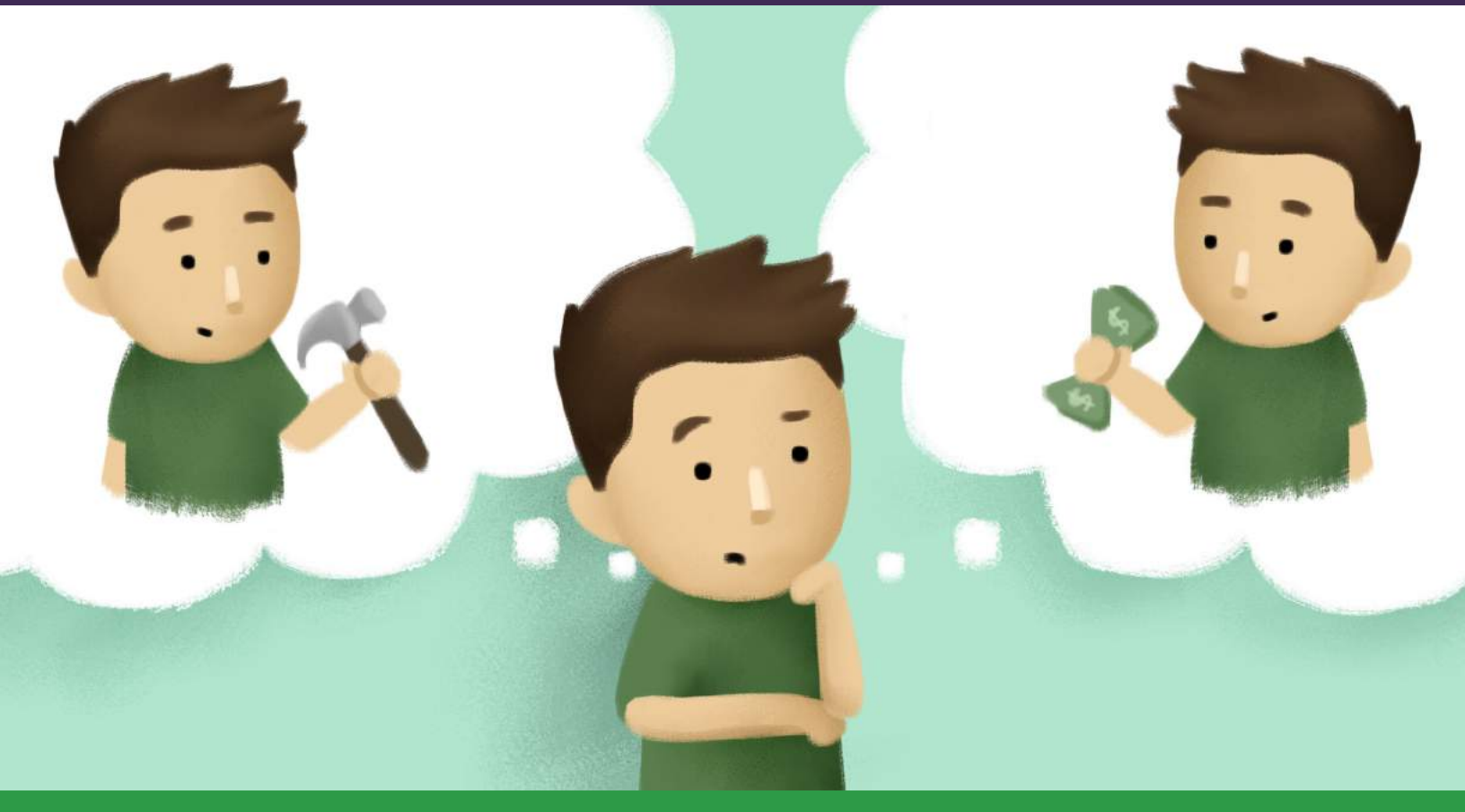




ButterCMS

MICROSERVICES FOR STARTUPS

Practical Advice For Teams Considering Microservices



Contributors



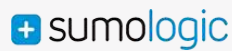
Tyler McMullen

CTO, FASTLY



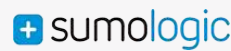
Christian Beedgen

CTO, SUMO LOGIC



Stefan Zier

CHIEF ARCHITECT, SUMO LOGIC



Julien Lemoine

CTO, ALGOLIA



Marco Palladino

CTO, KONG



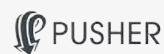
Chris McFadden

VP OF ENG, SPARKPOST



Sam Stagg

VP OF ENG, PUSHER



Oleksiy Kovyrin

HEAD OF SWIFTYPE, SRE, ELASTIC



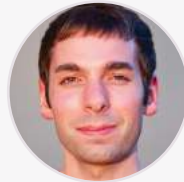
Russell Smith

CTO, RAINFOREST QA



David Strauss

CTO, PANTHEON



Stephen Blum

CTO, PUBNUB



Travis Reeder

CTO, IRON.IO



Steven Czerwinski

HEAD OF ENG, SCALYR



David Calavera

CTO, NETLIFY



Khash Sajadi

CEO, CLOUD 66



Zachary Crockett

CTO, PARTICLE





Slawomir Zabkiewicz

VP OF ENG, FILESTACK



Jean-Baptiste Aviat

CTO, SSCREEN



Shamasis Bhattacharya

VP OF ENG, POSTMAN



Joel Meador

CO CREATOR AT DOCRAPTOR



Mike Hu

HEAD OF ENG, LOGDNA



Darby Frey

CO-FOUNDER, LEAD HONESTLY



Isaac Mosquera

CTO, ARMORY



Nick Zadrozny

CEO AT BONSAI



Ben Curtis

CEO, HONEYBADGER



Jake Lumetta

CEO, BUTTERCMS



What's Inside

Learn how to determine proper service boundaries, break apart your monolith, and more.

CHAPTER 1

How Teams Get Microservices Wrong from the Start

There's a lot of ambiguity and debate about what microservices are.

CHAPTER 2

Should You Always Start with a Monolith?

Conventional wisdom says so but are there exceptions?

CHAPTER 3

Microservices Boundaries: Five Characteristics to Guide Your Design

Are your microservices too small? Too tightly coupled? This design guide can help.

CHAPTER 4

Five Microservices Testing Strategies for Startups

An in-depth look at strategies for testing your services.

CHAPTER 5

Should You Break Up Your Monolithic Application?

You've got a monolith and it's served you well but is it time to break it up?

CHAPTER 6

Breaking Up a Monolith: Kong Case Study

An inside look at how Kong tackled breaking up their monolith.

CHAPTER 7

Designing a Successful Microservices Engineering Culture

How freedom, responsibility, and team organization factor into successful microservices.

CHAPTER 8

Should You Build or Buy Microservices?

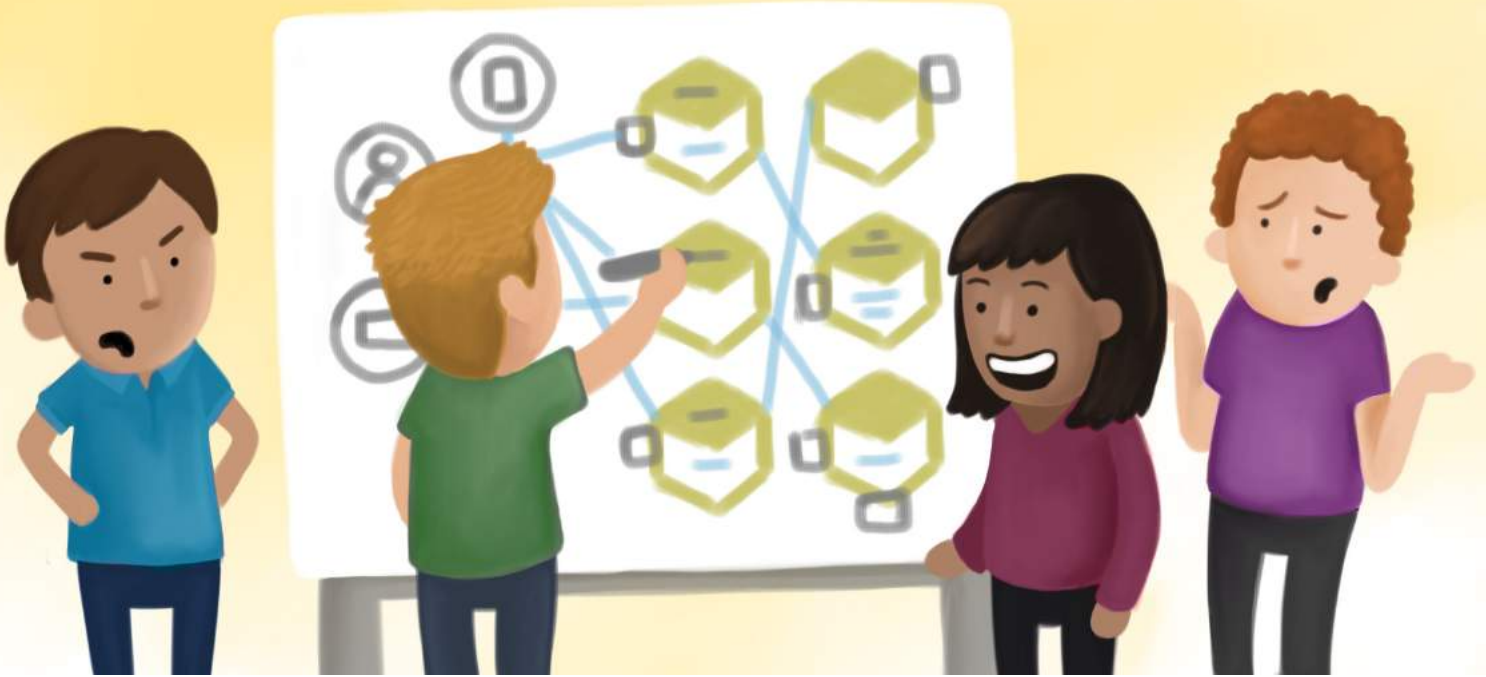
The microeconomics of microservices.



MICROSERVICES FOR STARTUPS // CHAPTER 1

How Teams Get Microservices Wrong from the Start

JAKE LUMETTA, CEO, ButterCMS
[@JAKELUMETTA](#)



There's a lot of ambiguity & debate about what microservices are.

When I first mentioned the concept of microservices to my engineering team, I was surprised by how quickly everyone researched the idea and then jumped to the conclusion that our monolith should be split up into tiny APIs that were each responsible for a model in our existing Rails application.

From the research I'd done, I knew that it was dangerous to build a bunch of microservices without careful consideration about size, boundaries, and other tradeoffs. But no one on my team seemed concerned about it.

So I probed my team with questions. Why so many services? Why not less? What were the tradeoffs?

What I found was that people on my team were jumping to conclusions based on shallow but dangerously firm notions of what a

microservice was. They held knowledge that microservices were small APIs pieced together to create whole systems. But they weren't aware of the intricate tradeoffs and design considerations that can mean the difference between success and failure.

They were pitching architectures with little ability to justify or reason about them. They had simply jumped to the conclusion of creating a bunch of APIs because that's what they thought microservices were supposed to be.

I wish I could say this was a unique occurrence, but I've worked on several teams where this has happened. If you're reading this, you probably have too.

Why does this happen? And what is a microservice, anyway?

Poorly named, ambiguously defined

For a profession that stresses the importance of naming things well, we've done ourselves a disservice with microservices. The problem is that there is nothing inherently "micro" about microservices. Microservices do not have to be small. Some are, but size is relative and there's no standard of unit of measure across organizations. A "small" service at one company might be one million lines of code while far less at another.

Some argue that microservices aren't a new thing at all and are rather a rebranding of Service Oriented Architectures, while others advocate for viewing microservices as an implementation of SOA similar to how Scrum is an implementation of Agile. This poor naming

is one of the reasons behind misconceptions and misapplication of microservices. Lots of developers, even those in 3-man teams, are jumping on the bandwagon of splitting up apps into far too many services without understanding what microservices really are.

When a popular, attractive-sounding term like "microservices" flies around, it's easy for the concepts behind it to get lost and distorted. Martin Fowler calls this Semantic Diffusion.

Semantic diffusion is essentially a succession of Chinese whispers where a different group of people from the originators of a term start talking about it without being careful about following the original definition. These people are listened to by a further group which then goes on to add their own distortions. After a few of these hand-offs it's easy to lose a lot of the key meaning of the term unless you make the point of going back to the originators.

The misconceptions don't just affect people who want to use microservices, they also stoke the fires of those that are dismayed by the industry hopping on the microservices bandwagon without deep understanding of its concepts.



Simon Brown
@simonbrown



I'll keep saying this ... if people can't build monoliths properly, microservices won't help. [#qconlondon](#) [#DesignThinking](#) [#Modularity](#)

5:49 AM - Mar 4, 2015

♡ 181 💬 322 people are talking about this



Most critique is not necessarily a critique of microservices themselves, but rather criticism of failed application of microservices coupled to (and likely caused by) a misunderstanding of microservices.

Without this knowledge, failures are attributed to the entire concept of microservices rather than the failure to use them successfully.

The Ideal Definition

There's a lot of ambiguity around what microservices are in part because no precise definition exists. Like Agile, microservices are a collection of broad concepts rather than concrete practices.

How exactly the term “microservice” was invented has been documented by Martin Fowler:

The term “microservice” was discussed at a workshop of software architects near Venice in May, 2011 to describe what the participants saw as a common architectural style that many of them had been recently exploring. In May 2012, the same group decided on “microservices” as the most appropriate name.

[Jame's Lewis's 2012 presentation at 33rd Degree in Krakow titled “Microservices - Java, the Unix Way”](#) presented these new ideas and described microservices as a way of develop software faster by dividing and conquering using Conway's law to structure teams.

Since then many others have continued to pioneer and advance the concepts around microservices including Fred George, Adrian Cockcroft (Netflix), Martin Fowler (ThoughtWorks), and Sam Newman (ThoughtWorks).

Today's leading definitions are fairly well aligned:

Microservices are small, autonomous services that work together.

—**Sam Newman**

Fine grained SOA architecture done the UNIX way.

—**James Lewis**

Loosely coupled service oriented architecture with bounded contexts.

—**Adrian Crockford**

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

[Martin Fowler and James Lewis](#)

The problem with these definitions is that while they are helpful for introducing the idea of microservices, but when it comes to time put them into practice, they are not very helpful. Using these definitions, how would you determine whether it makes more sense to have 10 tiny services versus five medium-sized ones?

In his book *Building Microservices*, Sam Newman explains that the right size is “Small enough and no smaller... We seem to have a very good sense of what is too big, and so it could be argued that once a piece of code no longer feels too big, it’s probably small enough.” Again, this doesn’t help you make a clear decision.

Ambiguous descriptions of microservices aren’t useful other than as an introduction. When it comes time to put microservices into practice, you need to find other ways to align your team.

Getting alignment on your team

How do you align your team when no precise definitions of microservices exist? The most important thing when talking about microservices on a team is to ensure that you are grounded in a common starting point.

But ambiguous definitions don’t help with this. It would be like trying to put Agile into practice without context for what you are trying to achieve, or an understanding of precise methodologies like Scrum.

On one team I worked on, we tried to not get hung up on definitions and instead, we first focused on defining the benefits we were trying to achieve by adopting microservices:

SHIPPING SOFTWARE FASTER

Our main application was a large codebase with several small teams of developers trying to build features for different purposes. This meant that every change had to try to satisfy the different groups. For example, a database change that was only serving one group would have to be reviewed and accepted by others that didn’t have as much context. This was tedious and slowed us down.

Having different groups of developers sharing the same codebase also meant that the code continually grew more complex in undeliberate ways. As the codebase grew larger, no one on the team could own it and make sure all the parts were organized and fit together optimally. This also made deploying a scary ordeal. A one-line change to our application still required the whole thing to be deployed in order to push out the change. Because deploying our large application was high-risk, our QA process grew and we deployed less.

With a microservices architecture, we hoped to be able to divide our code up so different teams of developers could fully own them. This would enable teams to innovate much more quickly without tedious design, review, and deployment processes. We also hoped that by having smaller codebases worked on by fewer developers, our codebases would be easier to develop, test, and keep well organized.

FLEXIBLY WITH TECHNOLOGY CHOICES

Our main large application was built with Ruby on Rails and a custom JavaScript framework with complex build processes.

Several parts of our application were hitting major performance issues that were difficult to fix and bringing down the rest of the application with it. We saw an opportunity to rewrite these parts of our application using a better approach, but inter-tangled our codebase was with the affected areas, this felt too big and costly to do.

At the same time, one of our front-end teams was wanting to pull away from our custom JavaScript framework and instead build product features with a newer framework like React. But mixing React into our existing application and complex front-end build process seemed expensive to configure.

As time went on, our teams grew frustrated with the feeling of being trapped in a big codebase that was too big and expensive to fix or replace.

By adopting microservices architecture, we hoped that keeping individual services smaller would mean that the cost to replace them with a better implementation would be much easier to manage. We also hoped to be able to pick the right tool for each job rather than having to go with a one-size-fits-all approach. We'd have the flexibility to use multiple technologies across our different applications as we saw fit. If a team wanted to use something other than Ruby for better performance, or switch from our custom JavaScript framework React, they could do so.

MICROSERVICES ARE NOT A FREE LUNCH

In addition to outlining the benefits we hoped to achieve, we also made sure we were being realistic about the costs and challenges associated with building and managing microservices.

Developing, hosting, and managing numerous services requires substantial overhead. A single monolith running on a few processes can easily translate to a couple dozen processes across a handful of services, requiring load balancers, messaging layers, and clustering for resiliency. Managing all this requires substantial skill and tooling.

Furthermore, microservices involve distributed systems which introduce a whole host of concerns such as network latency, fault tolerance, transactions, unreliable networks, and asynchronicity.

Setting your own path

Once we defined the benefits and costs of microservices, we could talk about architecture without falling into counterproductive debates

about who was doing microservices right or wrong. Instead of trying to find our way using others' descriptions or examples of microservices, we instead focused on the core problems we were trying to solve.

- *How would having more services help us ship software faster in the next 6-12 mos?*
- *Were there strong technical advantages to using a specific tool for a portion of our system?*
- *Did we foresee wanting to replace one of the systems with a more appropriate one down the line?*
- *How did we want to structure our teams around services as we hired more people?*
- *Was the productivity gain from having more services worth the foreseeable costs?*

In summary, here are the recommended steps for aligning your team that is jumping into microservices:

- 1 Learn about microservices while agreeing that there is no "right" definition. Discuss and memorialize your anticipated benefits and costs of adopting microservices
- 2 Avoid too eagerly hopping on the microservices bandwagon—be open to creative ideas and spirited debate about how best to architect your systems. Stay rooted in the benefits and costs you have previously identified.
- 3 Focus on making sure the team has a concretely defined set of common goals to work off of. It's more valuable to discuss and define what you'd like to achieve with microservices than it is to try and pin down what a microservice actually is.

Thanks to Darby Frey for providing feedback on this chapter.



MICROSERVICES FOR STARTUPS // CHAPTER 2

Should You Always Start with a Monolith?

JAKE LUMETTA, CEO, ButterCMS
[@JAKELUMETTA](#)



Conventional wisdom says start with a monolith but are there exceptions?

My good friend Darby Frey recently kicked off a greenfield project after assuming his new role as Sr. Platform Engineering Lead of Gamut. Despite starting out with monolith at his previous company Belly, he discovered that — in the right circumstances — starting with a monolith isn't always the best way to go.

"As one does, I allowed much of my thinking in my early days [at my new company] to be influenced by my previous company," Darby told me.

At Belly, Darby and his team broke down their monolith into a fairly large microservices

architecture. They managed to get it to a good place, but only after months of trials and tribulations migrating to microservices.

With this experience fresh in his mind, he approached his new project at Gamut a bit more cautious of microservices.

“I was firmly a member of Team Monolith. [I thought] let’s build a single application and just pull things apart later if we start to feel pain,” he said.

While this was a greenfield project, Darby’s team was small, and he had aggressive timelines, so on the surface, a monolith seemed like the obvious choice.

“[But with this new project], I was anxious to not repeat the mistakes of the past.”

And with that, he found himself faced with a decision we all struggle with, should we start with a monolith or microservices and how do we decide?

“I was firmly a member of Team Monolith. Let’s build a single application and just pull things apart later if we start to feel pain” —Darby Frey

Monolith Vs Microservices: Knowing The Difference

To decide between the two, we should first nail down what exactly we mean by “monolith” and “microservice.”

“System architectures lie on a spectrum.” Zachary Crockett, CTO of Particle told me during an interview.

“When discussing microservices, people tend to focus on one end of that spectrum: many tiny applications passing too many messages to each other. At the other end of the spectrum you have a giant monolith doing too many things. For any real system, there are many possible service oriented architectures between those two extremes,” he explained.

WHAT IS A MONOLITH?

A monolithic application is built as a single, unified unit. Often a monolith consists of three parts: a database, a client-side user

interface (consisting of HTML pages and/or JavaScript running in a browser), and a server-side application. The server-side application will handle HTTP requests, execute domain-specific logic, retrieve and update data from the database, and populate the HTML views to be sent to the browser.

Another characteristic of a monolith is that it’s often one massive code base. Server side application logic, front end client side logic, background jobs, etc, are all defined in the same code base. This means if developers want to make any changes or updates, they need to build and deploy the entire stack all at once.

Contrary to what you might think, a monolith isn’t a dated architecture that we need to leave in the past. In certain circumstances, a monolith is ideal. I spoke to Steven Czerwinski, Head of Engineering at Scaylr and former Google employee, to better understand this.

“Even though we had had these positive experiences of using microservices at Google, we [at Scaylr] went [for a monolith]

route because having one monolithic server means less work for us as two engineers,” he explained. This was back in the early beginnings of Scalyr.

In other words, because his team was small, a unified application was more manageable in comparison to splitting everything up into microservices.

MONOLITH PROS

- + **Fewer Cross-cutting Concerns:** A major advantage associated with monolithic architecture is that you only need to worry about cross-cutting concerns, such as logging or caching, for one application.
- + **Less Operational Overhead:** Focusing your finances on one application means that there’s only one application that you need to set up logging, monitoring and testing for. A monolith is also generally less complex to deploy since you aren’t organizing multiple deployments.
- + **Easier Testing:** With a monolith, automated tests are easier to setup and run because, once again, everything is under the same roof. With microservices, tests will need to accommodate for different applications on different runtime environments — which can get complex.
- + **Performance:** A monolith can also boast performance advantages in comparison to microservices. That’s often down to a monolith using local calls instead of an API call across a network.

MONOLITH CONS

- **Overly-tight Coupling:** While monoliths can help you avoid entanglement as previously mentioned, a monolith becomes more vulnerable to entanglement the

larger it grows. Because everything is so tightly coupled, isolation of services within the monolith becomes arduous, making life difficult when it comes to independent scaling or code maintenance.

- **Harder To Understand:** It’s common to find that monoliths are more difficult beasts to understand in comparison to microservices, a problem which rears its head when on-boarding new team members. This is sometimes a direct result of the tight coupling, as well as the fact that there may be dependencies and side-effects which are not obvious when you’re looking at a particular service or controller.

WHAT ARE MICROSERVICES?

As we discussed in Chapter One, [“How Teams Get Microservices Wrong From The Start”](#), the developer community have done themselves a disservice with the naming of this architecture.

The problem is that there is nothing inherently “micro” about microservices per se. While they tend to be smaller than the average monolith, they do not have to be tiny. Some are, but size is relative and there’s no standard of unit of measure across organizations.

At this point, it’s worth mentioning that — as you might have gathered from the slightly varying definitions given above — there is no industry consensus of what exactly microservices are. Nevertheless, here’s my take on the definition of microservices:

Microservice architecture refers to the concept of developing a single application as a suite of small services, in contrast to developing them as one, large ‘monolith’.

Each of those broken-up, individualized services run on their own process, communicating with lightweight mechanisms,

often an HTTP resource API. Fully-fledged microservices are independently deployable, and yet can work in tandem when necessary.

MICROSERVICES PROS

- + **Better Organization:** Microservice architectures are typically better organized, since each microservice has a very specific job, and is not concerned with the jobs of other components.
- + **Decoupled:** Decoupled services are also easier to recompose and reconfigure to serve the purposes of different apps (for example, serving both the web clients and public API). They also allow for fast, independent delivery of individual parts within a larger, integrated system.
- + **Performance:** Under the right circumstances, microservices can also have performance advantages depending on how they're organized because it's possible to isolate hot services and scale them independent of the rest of the app.
- + **Fewer Mistakes:** Microservices enable parallel development by establishing a hard-to-cross boundary between different parts of your system. By doing this, you make it hard – or at least harder – to do the wrong thing: Namely, connecting parts that shouldn't be connected, and coupling too tightly those that need to be connected.

MICROSERVICES CONS

- **Cross-cutting Concerns Across Each Service:** As you're building a new microservice architecture, you're likely to discover lots of cross-cutting concerns that you did not anticipate at design time. You'll either need to incur the overhead of separate modules for each cross-cutting concern (i.e. testing), or encapsulate

cross-cutting concerns in another service layer that all traffic gets routed through. Eventually, even monolithic architectures tend to route traffic through an outer service layer for cross-cutting concerns, but with a monolithic architecture, it's possible to delay the cost of that work until the project is much more mature.

- **Higher Operational Overhead:** Microservices are frequently deployed on their own virtual machines or containers, causing a proliferation of VM wrangling work. These tasks are frequently automated with container fleet management tools.

Every time we consider introducing a new service, we have to consider the operational cost of doing so. Each new service adds to the complexity of the infrastructure and makes it a bit harder to reason about service relationships within the system.

—Oleksiy Kovyrin, Head of Swifttype SRE, Elastic

Monolith Vs Microservices: How To Decide

Based on interviewing CTO's that have made this decision, there are several key considerations you should make when deciding for your own organization.

ARE YOU IN FAMILIAR TERRITORY?

Darby and his team at Gamut were able to delve directly into Microservices because he had experience with eCommerce platforms, and his company had a wealth of knowledge concerning the needs and demands of

their customers. If he was traveling down an unknown path on the other hand, a monolith may have actually been the safer option.

Similarly, often startups are born out of pains experienced at previous companies. In those scenarios sometimes it's quite clear scaling is going to be a primary requirement, especially in infrastructure based services like cloud log management.

IS YOUR TEAM PREPARED?

Does your team have experience with microservices? What if you quadruple the size of your team within the next year, are microservices ideal for that situation? Evaluating these dimensions of your team is crucial to the success of your project.

Julien Lemoine, CTO at Algolia, chimed in on this point:

"We have always started with a microservices approach. The main goal was to be able to use different technology to build our service, for two big reasons:

- 1 We want to use the best tool for each service. Our search API is highly optimized at the lowest level and C++ is the perfect language for that. That said, using C++ for everything is a waste of productivity, especially to build a dashboard!
- 2 We want the best talents and using only one technology would limit our options. This is why we have different languages in the company, Go is less perfect than C++ when you want to optimize everything at the millisecond level but is the perfect language when performance is still key (processing of logs where we process several terabytes of logs per day, using ruby or python would be a waste of CPU)"

If your team is prepared, starting with microservices is wise as it allows you to get used to the rhythm of developing in a microservice environment, right from the start.

HOW'S YOUR INFRASTRUCTURE?

In reality, you're going to need cloud-based infrastructure to make microservices work for your project.

"[Previously], you would want to start with a monolith because you wanted to deploy one database server. The idea of having to set up a database server for every single microservice and then scale out was a mammoth task. Only a huge, tech-savvy organization could do that," David Strauss, CTO of Pantheon explained to me.

"Whereas today with services like Google Cloud and Amazon AWS, you have many options for deploying tiny things without needing to own the persistence layer for each one."

EVALUATE THE BUSINESS RISK

You may think microservices is the "right" way to go as a tech-savvy startup with high ambitions. But microservices pose a business risk. David Strauss explained:

"A lot of teams overbuild their project initially; everyone wants to think their startup will be the next unicorn and that they should, therefore, build everything with microservices or some other hyper-scalable infrastructure. But that's usually wrong, almost all the time," he said.

One example of this from his early days at Pantheon was a system that was limited to a single VM. They thought it would be a month or two until they'd be forced to scale it. It ended up being over a year -- and they ended up scaling it a completely different way than they had anticipated.

He went on to say that, in these cases, the areas that you thought you needed to scale are probably not the parts that will need to scale first, and that results in misplaced effort even for the systems that will need to scale.

Consider Your Context

Based on interviewing over a dozen CTO's, we've been able to distill some key insights that can help you in making this decision. Start by considering your own context.

WHEN TO START WITH A MONOLITH

Here are some scenarios that indicate that you should start your next project using monolithic architecture.

- **Your Team Is At Founding Stage:** Your team is small, between 2-5 members, and is thus unable to tackle a broader and high-overhead microservices architecture.
- **You're Building An Unproven Product or Proof of Concept:** Are you building an unproven product in the market? If it's a new idea, it's likely going to pivot and evolve over time, so a monolith is ideal to allow for rapid product iteration. Same applies to a proof of concept where your goal is just to learn as much as possible as quickly as possible, even if you end up throwing it away.
- **You Have No Microservices Experience:** If your team has no prior experience with microservices, unless you can justify taking the risk of learning "on the fly" at such an early stage, it's likely another sign you should stick to a monolith to start.

WHEN TO START WITH MICROSERVICES

Here are some scenarios that indicate that you should start your next project using microservices:

- **You Require Quick, Independent Service Delivery:** If it's snappy, isolated service delivery that you need, microservices are your best bet. However, depending on the size of your team, it can take some time before you see any service delivery gains versus starting with monolith.
- **A Piece of Your Platform Needs to Be Extremely Efficient:** If your business is doing intensive processing of petabytes of log volume, you'll likely want to build that service out in a very efficient language (i.e. C++) while your user dashboard may be built in Ruby on Rails.
- **You Plan To Scale Your Team:** Starting with microservices gets your team used to developing in separate small teams from the beginning, and having teams separated by service boundaries makes scaling your development organization easier.

Do not attempt to take on microservices just because other engineering teams are having success with them. Your own context, evaluated against the above considerations, is the key to deciding if you should start with monolith or microservices.

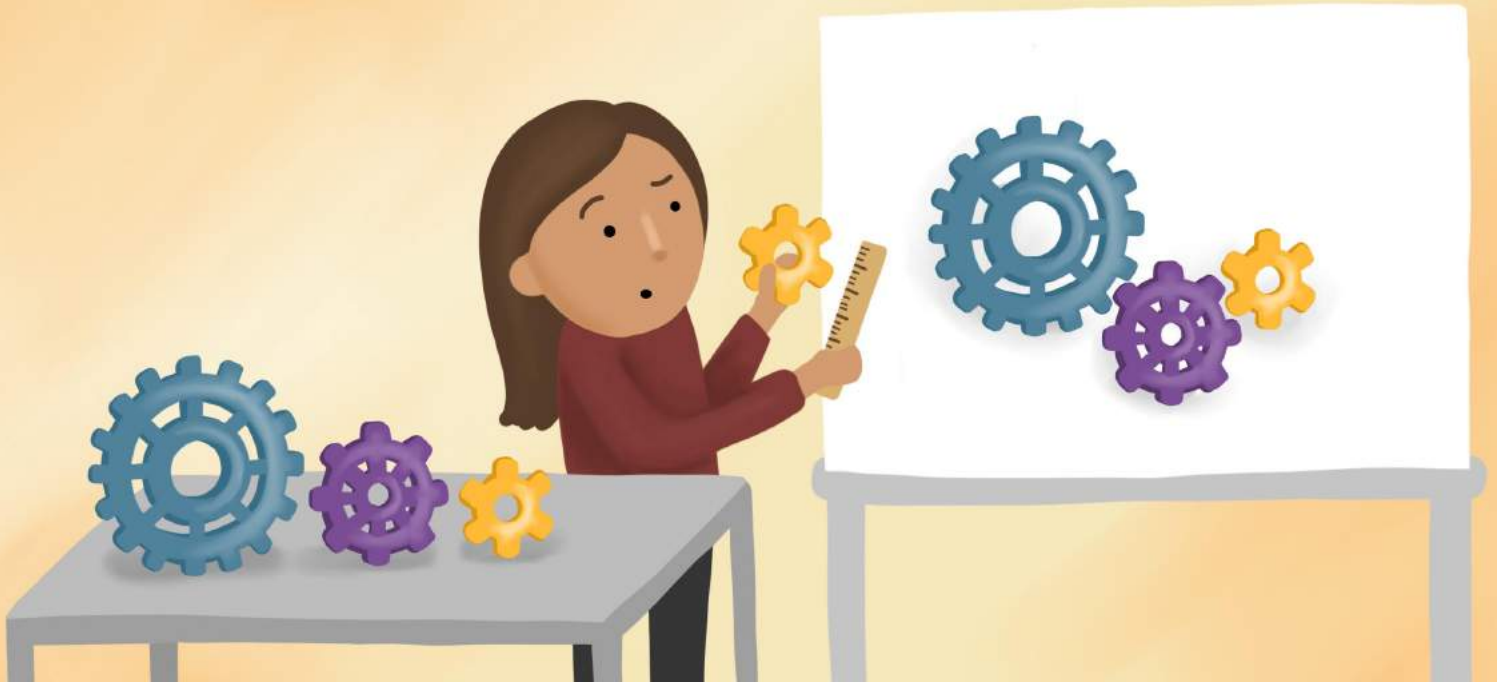
Thanks to Darby Frey, David Strauss, Steven Czwierinski, and Julien Lemoine, Zachary Crockett and Oleksiy Kovyrin for reviewing and contributing to this chapter.



MICROSERVICES FOR STARTUPS // CHAPTER 3

Microservice Boundaries: Five Characteristics to Guide Your Design

JAKE LUMETTA, CEO, ButterCMS
[@JAKELUMETTA](#)



Are your microservices too small? Too tightly coupled? This design guide can help.

In the early days of SparkPost Chris McFadden, VP of Engineering at SparkPost, and his team had to solve a problem that every SaaS business has: they needed to provide basic services like authentication, account management, and billing.

The core problem, of course, wasn't how to charge their users money. It was how to design their user account microservices to support everything that goes along with that problem domain: user accounts, API keys, and authentication, business accounts, billing, etc.

To tackle this they created two microservices: a Users API and an Accounts API. The Users

API would handle user accounts, API keys, authentication and the Accounts API would handle all of the billing related logic. A very logical separation, but before long, they spotted a problem.

"We had one service that was called the User API, and we had another one called the Account API. But the problem was that

they were actually having several calls back and forth between them. So you would do something in accounts and have call and endpoint in users or vice versa,” he continued.

The two services were too tightly coupled.

“We realized that in most cases, you really don’t want to have one service calling another service in a sometimes circular way. That’s generally not a good idea,” Chris explained.

I think this scenario will sound familiar to any developer who has ever tackled this design problem. I’ve made this mistake numerous times in the past. The question is, how do we avoid these microservice design pitfalls and what patterns should we look for? Read on to find out.

The Importance of Microservice Boundaries

When McFadden and I spoke further, he highlighted one of the major challenges when it comes to creating a new system with a microservice architecture. It came about when I mentioned that one of the [core benefits of developing new systems with microservices](#) is that the architecture allows developers to build and modify individual components independently — but problems can arise when it comes to minimizing the number of callbacks between each API. The solution according to McFadden, is to apply the appropriate service boundaries.

But in contrast to the sometimes difficult-to-grasp and abstract concept of domain driven design (DDD) — a framework for microservices — I’ll be as practical as I can in this chapter as I discuss the need for well defined microservice boundaries with some of our industry’s top CTOs.

Avoid Arbitrary Rules

When designing and creating a microservice, don’t fall into the trap of using arbitrary rules. If you read enough advice, you’ll come across some of the rules below. While appealing, these are not proper ways to determine boundaries for microservices.

1 “A MICROSERVICE SHOULD HAVE X LINES OF CODE”

Let’s get one thing straight; there are no limitations on how many lines of code there are in a microservice. A microservice doesn’t suddenly become a monolith just because you write a few lines of extra code. The key is ensuring there is high cohesion for the code within a service (more on this later).

2 “TURN EACH FUNCTION INTO A MICROSERVICE”

If a function that computes something based on three input values, and returns a result, is that a good candidate for a microservice? Should it be a separately deployable application of its own? This really depends on what the function is and how it serves to the entire system.

Other arbitrary rules include those that don’t take into account your entire context such as the team’s experience, DevOps capacity, what the service is doing and availability needs of the data.

Characteristics of a Well-designed Service

If you’ve read about microservices, you’ve no doubt come across advice on what makes

a well-designed service. Simply put: high cohesion and loose coupling. There are many articles on these concepts to review if you're not familiar with them. And while sound advice, these concepts are quite abstract.

I've spoken with dozens of CTO's on this topic to learn from them on how they've drawn their microservice boundaries, and I've distilled down some of the underlying characteristics for you below.

CHARACTERISTIC #1: IT DOESN'T SHARE DATABASE TABLES WITH ANOTHER SERVICE

As we saw in Chris McFadden's case mentioned earlier in this chapter, when it comes to designing a microservices if you have multiple services referencing the same table, that's a red flag as it likely means your DB is a source of coupling.

Each service should have its own tables [and] should never share database tables.
— Darby Frey, Co-founder of Lead Honestly

It is really about how the service relates to the data, which is exactly what Oleksiy Kovrin, Head of Swiftype SRE, Elastic, told me:

"One of the main foundational principles we use when developing new services is that they should not cross database boundaries. Each service should rely on its own set of underlying data stores. This allows us to centralize access controls, audit logging, caching logic, et cetera," he said.

Kovrin went on to explain that if a subset of your database tables, "have no or very little connections to the rest of the dataset, it is a strong signal that component could be isolated into a separate API or a separate service."

[Sam Newman](#) illustrates this scenario well and provides a couple of solutions.

You can [split the table apart \(27:41\)](#) or you can [reify a new service \(26:00\)](#).

Darby Frey, Co-founder of Lead Honestly, echoed this sentiment by telling me that, "each service should have its own tables [and] should never share database tables."

CHARACTERISTIC #2: IT HAS A MINIMAL AMOUNT OF DATABASE TABLES

As mentioned in [Chapter 1](#), the ideal size of a microservice is to be small enough, but no smaller. And the same goes for the number of database tables per service.

Steven Czerwinski, Head of Engineering, Scaylr, explained to me during an interview that the sweet spot for Scaylr is, "one or two database tables for a service."

"We have a suppression microservices, and it handles, keeps track of, millions and billions of entries around suppressions but it's all very focused just around suppression so there's really only one or two tables there.

The same goes for other services like webhooks,” explained Chris McFadden.

CHARACTERISTIC #3: IT’S THOUGHTFULLY STATEFUL OR STATELESS

When designing your microservice, you need to ask yourself whether it requires access to a database or is it going to be a stateless service processing terabytes of data like emails or logs.

We define the boundaries of a service by defining its input and output. Sometimes a service is a network API but it can also be a process consuming files and producing records in a database (this is the case of our log processing service).

—Julien Lemoine

Be clear about this upfront and it will lead to a better designed service.

CHARACTERISTIC #4: ITS DATA AVAILABILITY NEEDS ARE ACCOUNTED FOR

When designing a microservice, you need to keep in mind what services will rely on this new service and what’s the system-wide impact if that data becomes unavailable. Taking that into account allows you properly design data backup and recovery systems for this service.

When speaking to Steven Czerwinski, he mentioned their critical customer row space mapping data is replicated and separated in different ways due to its importance.

“Whereas the per shard information, that’s in its own little partition. It sucks if it goes down because that portion of the customer population is not going to have their logs available, but it’s only impacting 5 percent of

the customers rather than 100 percent of the customers,” Czerwinski explained.

CHARACTERISTIC #5: IT’S A SINGLE SOURCE OF TRUTH

The final characteristic to keep in mind is to design a service to be the single source of truth for something in your system.

To give you an example, when you order something from an eCommerce site, an order ID is generated. This order ID can be used by other services to query an Order service for complete information about the order. Using the pub/sub concept, the data that is passed around between services should either be the order ID, not the attributes/information of the order itself. Only the Order service has complete information and is the single source of truth for a given order.

Considerations for Larger Teams

For larger organizations, where entire teams can be dedicated to owning a service, organizational consideration comes into play when determining service boundaries. And there are two considerations to keep in mind: independent release schedule and different uptime importance.

“The most successful implementation of microservices we’ve seen is either based on a software design principle like domain-driven design for example, and service-oriented architecture or the ones that reflect an organizational approach,” said Khash Sajadi, CEO of Cloud66.

“So [for the] payments team,” Sajadi continued, “they have the payment service or credit card validation service and that’s the service

they provide to the outside world. So it's not necessarily anything about software. It's mostly about the business unit [that] provides one more service to the outside world."

Amazon is a perfect example of a large organization with multiple teams. As mentioned in an article published in [API Evangelist](#), Jeff Bezos issued a mandate to all employees informing them that every team within the company had to communicate via API. Anyone who didn't would be fired.

This way, all the data and functionality was exposed through the interface. Bezos also managed to get every team to decouple, define what their resources are, and make them available through the API. Amazon was

building a system from the ground up. This allows every team within the company to become a partner of one another.

I spoke to Travis Reeder, CTO of Iron.io, about Bezos' internal initiative.

"Jeff Bezos mandated that all teams had to build API's to communicate with other teams. He's also the guy who came up with the 'two pizza' rule; a team shouldn't be larger than what two pizzas can feed," he said.

"I think the same could apply here: whatever a small team can develop, manage and be productive with. If it starts to get unwieldy or starts to slow down, it's probably getting too big," Reeder told me.

[Jeff Bezos] came up with the 'two pizza' rule — a team shouldn't be larger than what two pizzas can feed.

—Travis Reeder, CTO of Iron.io

How to Tell If a Service Is Too Small, or Not Properly Defined

During the testing and implementation phase of your microservice system, there are a number of indicators to keep in mind.

The first indicator to look out for is any over-reliance between services. If two services are constantly calling back to one another, then that's a strong indication of coupling and a signal that they might be better off combined into one service.

Going back to the example Chris McFadden shared at the beginning of this chapter where he had two API services, accounts and users, that were constantly communicating with one another, McFadden came up with an idea to merge the services and decided to call it the Accuser's API. This turned out to be a fruitful strategy:

"What we started doing was eliminating these links [which were the] internal API calls between them. It's helped simplify the code." McFadden informed me.

The second is if the overhead of setting up the service outweighs the benefit of having it be independent.

Darby Frey explained, “Every app needs to have its logs aggregated somewhere and needs to be monitored. You need to set up alerting for it. You need to have standard operating procedures and run books for when things break. You have to manage SSH access to that thing. There’s a huge foundation of things that have to exist in order for an app to just run.”

Consider these characteristics

Designing microservices can often feel more like an art than a science. As an engineer, that doesn’t sit well with my left-brain. There’s lots of general advice out there but at times it can be a bit too abstract so let’s quickly recap:

- 1 It doesn’t share database tables with another service
- 2 It has a minimal amount of database tables

- 3 It’s thoughtfully stateful or stateless
- 4 Its data availability needs are accounted for
- 5 It’s a single source of truth

These are 5 specific characteristics to look for when designing your next set of microservices. So the next time you’re tasked with drawing the boundaries for new services like the Users and Accounts example mentioned at the beginning, I hope referring back to these helps make that task much easier.

Thanks to Chris McFadden, Darby Frey, Steven Czerwinski, Julien Lemoine, Oleksiy Kovyrin, Travis Reeder, and Khash Sajadi for reviewing and contributing to this chapter.



MICROSERVICES FOR STARTUPS // CHAPTER 4

Five Microservice Testing Strategies for Startups

JAKE LUMETTA, CEO, ButterCMS
[@JAKELUMETTA](#)



Testing microservices is hard but here's five ways your team can approach it.

Testing microservices is hard. More specifically, end-to-end testing is hard, and that's something we'll discuss in greater detail in this chapter.

But first, a quick story.

I learned just how hard microservice testing could be when I first dived into a tech stack with seven separate microservices, each with its own code base, dependency management, feature branches, and database schema — which also happened to have a unique set of migrations.

Talk about hectic.

The approach we took was to run everything locally. So that meant, at any given point in time that I want to run end-to-end tests, I would need to go through the following five steps for each of the seven microservices:

- 1 **Ensure I was on the correct code branch (either master or feature_xyz)**
- 2 **Pull down the latest code for that branch**
- 3 **Ensure all dependencies were up to date**
- 4 **Run any new database migrations**
- 5 **Start the service**

This was just a baseline requirement to enable tests to be run. Often times, I would forget to perform one of these steps for a service and spend 10-15 minutes debugging the issue. Once I finally had all the services happy and running, I could then begin to kick off the test suite. This experience sure made me miss the days of testing one big monolith.

So yes, I discovered that end-to-end microservice testing is hard — and gets exponentially harder with each new service you introduce. But fear not, because there are ways to make testing microservices easier. I've spoken to several CTOs about how they came up with their own creative ways of tackling this complex issue.

Common microservice testing methods

First, let's review the types of testing strategies to see where "end-to-end" testing falls on the microservice testing spectrum.

UNIT TESTING

A microservice may be smaller by definition, but with unit testing, you can go even more granular. A unit test focuses on the smallest part of a testable software to ascertain whether that component works as it should.

Renowned software engineer, author and international speaker Martin Fowler, [breaks unit testing down](#) into two categories:

- 1 **Sociable unit testing:** This unit testing method tests the behavior of modules by observing changes in their state.
- 2 **Solitary unit testing:** This method focuses on the interactions and collaborations between an object and its dependencies, which are replaced by test doubles.

While these unit testing strategies are distinct, Fowler puts forth that they aren't competing — they can be used in tandem to solve different testing problems.

During a discussion with David Strauss, CTO of Pantheon, he told me that, "The opportunity is that Microservices are very straightforward to actually do unit testing on."

Isaac Mosquera of Armory concurred, informing me that Armory "instructs customers to focus on unit testing."

INTEGRATION TESTING

With integration tests, you're doing what it sounds like you're doing: testing the communication paths and interactions between components to detect issues. [According to Fowler](#), an integration test, "exercises communication paths through the subsystem to check for any incorrect assumptions each module has about how to interact with its peers."

An integration test usually tests the interactions between the microservice and external services, like another microservice or datastore.

Pawel Ledwoń, Platform Lead, Pusher, informed me that his team, “lean[s] towards integration testing. Unit tests are still useful for some abstractions, but for user-facing features they are difficult to mock or skip important parts of the system.”

Sumo Logic’s Chief Architect Stefan Zier also revealed that they “invest fairly heavily into integration testing.”

However, not everybody I spoke to was a fan of the process. Mosquera’s take on the subject of integration testing, for example, is well worth noting:

“Integration testing is very error prone and costly, in terms of man-hours. The ROI just isn’t there. Each individual integration test brings small marginal coverage of use cases,” he told me.

“If you consider all the code path combinations to your application coupled with another application’s code paths the number of tests that need to be written easily explode to a number that is unachievable. Instead we instruct our customers to focus on unit test coverage, a handful of integration tests that will demonstrate total failure to key areas of the application,” Mosquera added.

Food for thought indeed!

Integration testing is very error prone and costly, in terms of man hours. The ROI just isn’t there.
—Isaac Mosquera, Armory

END-TO-END TESTING

Last but not least is end-to-end testing, which — as previously mentioned — can be a difficult task. That’s because it involves testing every moving part of the microservice, ensuring that it can achieve the goals you built it for.

[Fowler wrote](#) that, “end-to-end tests may also have to account for asynchrony in the system, whether in the GUI or due to asynchronous backend processes between the services.” He goes on to explain how these factors can result in “flakiness, excessive test runtime and additional cost of maintenance of the test suite.”

The best advice I can give when it comes to end-to-end testing is to limit the amount of times you attempt it per service. A healthy

balance between the other microservice testing strategies mentioned — like unit testing and integration testing — will help you weed out smaller issues. An end-to-end test is larger by definition, takes more time and can be far easier to get wrong. To keep your costs low and avoid time-sink, stick to end-to-end testing when all other means of testing have been exhausted, and as a final stamp of quality assurance.

Testing microservices vs testing monoliths

I’ve already gone deep on the [monolith vs microservice debate](#), but what are the differences between the two when it comes to testing in particular?

Right off the bat, microservices require extra steps like managing multiple repositories and branches, each with their own database schemas. But the challenges can run deeper than that.

Here are a few key challenges associated with testing microservices:

- **Availability:** Since different teams may be managing their own microservices, securing the availability of a microservice (or, worse yet, trying to find a time when all microservices are available at once), is tough.
- **Fragmented and holistic testing:** Microservices are built to work alone, and together with other loosely coupled services. That means developers need to test every component in isolation, as well as testing everything together.
- **Knowledge gaps:** Particularly with integration testing (which we will address later in this chapter), whoever conducts the tests will need to have a strong understanding of each service in order to write test cases effectively.

According to Oleksiy Kovyryn, Head of Swifttype SRE at Elastic, “one important issue we have to keep in mind while working with microservices is API stability and API versioning. To avoid breaking applications depending on a service, we need to make sure we have a solid set of integration tests for microservice APIs and, in case of a breaking change, we have to provide a backwards-compatible way for clients to migrate to a new version at their own pace to avoid large cross-service API change rollouts.”

Stefan Zier, Chief Architect at Sumo Logic, also reiterated to me that microservice testing is indeed “very, very difficult.”

“Especially once you go towards more continuous deployment. We’ve invested and

continue to invest fairly heavily into integration testing, unit testing, and would do a lot more if we had the people to do it,” Zier told me. With that being said, he admitted that, at certain stages, when Sumo Logic wants to test their services holistically, “more or less [the] entire company [becomes] a quality assurance team in a sense.”

Five microservice testing strategies for startups

Yes, testing microservices can be difficult, but [given all the benefits of microservices](#), foregoing them because of testing challenges isn’t the right path. To tackle these challenges, I got insight from several CTOs and distilled five strategies they used to successfully approach testing microservices.

1. THE DOCUMENTATION-FIRST STRATEGY

Chris McFadden, VP of Engineering Sparkpost, summarized the documentation-first strategy quite nicely during our discussion:

“We follow a documentation first approach so all of our documentation is in markdown in GitHub. Our API documents are open source, so it’s all public. Then, what we’ll do is before anybody writes any API changes or either a new API or changes to an API, they will update the documentation first, have that change reviewed to make sure that it conforms with our API conventions and standards which are all documented, and make sure that there’s no breaking change introduced here. Make sure it conforms with our naming conventions and so forth as well.”

—Chris McFadden, VP of Engineering, Sparkpost

If you're willing to go one step further, you could dabble in API contract testing, which — as previously mentioned — involves writing and running tests that ensure the explicit and implicit contract of a microservice works as it should.

2. THE FULL STACK IN-A-BOX STRATEGY

The full stack in-a-box strategy entails replicating a cloud environment locally and testing everything all in one vagrant instance (“\$ vagrant up”). The problem? It's extremely tricky, as software engineer Cindy Sridharan of Imgix explained in a [blog post](#):

“I've had first hand experience with this fallacy at a previous company I worked at where we tried to spin up our entire stack in a [local] vagrant box. The idea, as you might imagine, was that a simple vagrant up should enable any engineer in the company (even frontend and mobile developers) to be able to spin up the stack in its entirety on their laptops,” she begins.

Sridharan goes on to detail how the company only had two microservices, a event based API server and some asynchronous Python background workers. A relatively simple setup, by all means.

“I remember spending my entire first week at this company trying to successfully spin up the VM locally only to run into a plethora of errors. Finally, at around 4:00pm on the Friday of my first week, I'd successfully managed to get the Vagrant setup working and had all the tests passing locally. I remember feeling incredibly exhausted,” she narrated.

Despite her best efforts to document the obstacles she ran into — and how she got over them — Sridharan revealed that the next engineer her company hired ran into their own share of issues that couldn't be replicated on another laptop. I did say it was tricky.

Plus, Stefan Zier, Chief Architect at Sumo Logic, explained to me that — on top of being difficult

to pull off — this localized testing strategy simply doesn't scale:

“[With] a local deployment, we run most of the services there so you get a fully running system and that's now stretching even the 16GB RAM machines quite hard. So that doesn't really scale,” he said.

3. THE AWS TESTING STRATEGY

This third strategy involves spinning up an Amazon Web Services (AWS) infrastructure for each engineer to deploy and run tests on. This is a more scalable approach to the full stack in-a-box strategy discussed above.

Zier called this a “personal deployment [strategy], where everyone here has their own AWS account.”

“You can push the code that's on your laptop up into AWS in about ten minutes and just run it in like a real system,” he said.

4. THE SHARED TESTING INSTANCES STRATEGY

I like to think of this fourth strategy as a hybrid between full stack in-a-box and AWS testing. That's because it involves developers working from their own, local station, while leveraging a separate, shared instance of a microservice to point their local environment at during testing.

Steven Czerwinski, Head of Engineering, Scaylr, explained how this can work in practice:

“Some of [our] developers run a separate instance of a microservice just to be used for testing local builds. So imagine you're a developer, you're developing on your local workstation, and you don't have easy way of launching the image parser. However, your local builder would just point to a test image parser that's running in the Google infrastructure,” he said.

“Similarly,” Czerwinski continued, “we’ve been talking about doing that for the front end developers. Hiding the database layer between a service so that people don’t have to run their own kind of monolithic server, the database server, when they’re testing UI features. They want to be able to rely on an external one that can be updated pretty frequently and doesn’t have production or staging data or anything like that.”

5. THE STUBBED SERVICES STRATEGY

And finally, we have the stubbed services testing strategy.

Zier laid out Sumo Logic’s approach to stubbed service testing by telling me how, “stubbing let’s you write these marks or ‘stubs’ of microservices that behave as if they were the right service and they advertise themselves in our service discovery as if they were real service, but they’re just a dummy imitation,” he explained.

For example, testing a service may necessitate that the service becomes aware that a user carries out a set of tasks. With stubbed services, you can pretend that user (and his tasks) have taken place without the usual complexities that come with that. Obviously, this approach is a lot more lightweight compared to running services in their totality.

Tools to help you test microservices

Here’s a list of tools that have benefited me during my own microservice testing experiences, bolstered by some recommendations from the pool of CTOs and senior developers who contributed to this eBook:

- [Hoverfly](#): simulate API latency and failures.

- [Vagrant](#): build and maintain portable virtual software development environments
- [VCR](#): a unit testing tool
- [Pact](#): frameworks consumer-driven contracts testing.
- [Apiary](#): API documentation tool
- [API Blueprint](#): design and prototype APIs
- [Swagger](#): design and prototype APIs

Microservice testing: difficult, but worth it

Testing your microservices won’t be a walk in the park, but the additional work is worth the benefits of having a microservice architecture.

Plus, the microservice testing strategies adopted by the likes of Sumo Logic and Scaylr, should help smooth the process out. Here’s a quick recap of those strategies:

- 1 The documentation-first strategy
- 2 The full stack in-a-box strategy
- 3 The AWS testing strategy
- 4 The shared testing instances strategy
- 5 The stubbed service strategy

Naturally, you may need to tweak each strategy to match your unique circumstances, but with some good old fashioned trial and error, your microservice testing strategy should come into its own.

Thanks to David Strauss, Steven Czerwinski, Oleksiy Kovyrin and Stefan Zier, Paweł Ledwoń, Chris McFadden and Isaac Mosquera for reviewing and contributing to this chapter.



MICROSERVICES FOR STARTUPS // CHAPTER 5

Should You Break Up Your Monolithic Application?

MARCO PALLADINO, CTO, [KONG](#)
[@THEFOSK](#)



You've got a monolith and it's served you well but is it time to break it up?

Microservices and monolithic architectures both have their pros and cons, and choosing which architecture to build needs to be a thoughtful decision. Microservice architectures have recently been in the spotlight and most large organizations I have been personally working with are either considering - or in the process - of transitioning to microservices, a trend that's happening horizontally across every industry. Despite what everybody else is doing and the noise everybody is making around microservices, we need to zoom out and look at our long term goals and objectively understand if it makes sense to break up the monolith or not.

A microservice-oriented application does not make it easy to quickly build something and validate it in the market. In fact it would be quite the opposite; the amount of concerns and moving parts that microservices bring to the fight will certainly slow you down initially.

If the plan is to create a prototype or validate the market for a new product, monolithic applications are still a very good choice (as long as we consider that in the long term we may have to revisit our architecture).

On the other side if we do already have an application running and we are firefighting with common issues related to an increase in size of the codebase (or the team), and if deployments are becoming increasingly painful, then microservices can fix those problems. Microservices effectively replace function calls within the monolith with network calls across different services, increasing the networking requirements and complexity of the application. Some of the new topics you will have to start thinking of are - for instance - the performance of the network, the security and reliability of all of those network calls your architecture will start making and the visibility and monitoring across the entire system. If your team currently doesn't have a good knowledge of APIs, networking and perhaps containers (Docker, Kubernetes), then you will have to prepare the transition by making a few strategic hires that can bring this knowledge in house.

Containers and orchestration tools are not - from a technical standpoint - a requirement for microservice-oriented applications, but they can certainly help in decoupling and scaling each one of the different services you will be creating and deploying. Orchestration platforms like Kubernetes also provide some useful glue functionality out of the box to make a sense of your architecture, providing tooling that otherwise you will have to deploy - and sometimes build - by yourself (for example service discovery, network management, versioning, etc).

While a few years ago being locked-in with a cloud provider was generally accepted, I am seeing more and more enterprises reversing this trend and making technical decisions that would allow them to move their infrastructure to different cloud providers if such necessity should arise. These enterprises are thinking strategically about their architecture: when you are planning your architecture and execution

on your vision down the road, with all the tooling available today there is really no excuse to be locked in with a specific vendor. Tooling like containers and Kubernetes allow your application to be replicated and deployed across different cloud providers, on-premise, or locally for development.

If you are transitioning an existing monolithic codebase to microservice without an appropriate test suite in place already, then the answer is no. You should already have - or you should start building - a comprehensive suite of both unit and integration tests before approaching any refactoring.

As we will learn in this chapter it doesn't have to be an either/or choice: you can still keep some of your codebase as it is, and only transition the most painful and demanding business logic to a microservice-oriented architecture. Obviously you will have to glue them together in order to share state and functionality.

Finally, the decision of fundamentally transitioning from one architecture to another is strategic and not tactical, therefore it needs to be well understood by both the teams and the management. Going to microservices is not just a technology transition, but an organizational transition in the way the teams are created and managed, and in the way those teams are working and collaborating with each other. It's not just a matter of adopting new frameworks or new tools, it's a revolution that fundamentally changes every aspect of the software lifecycle. The transition is also not going to happen overnight and the teams need the full blessing of the leadership and their understanding of what the future, and the transition itself, will look like. Patience is a key factor while undertaking such a massive project.

A technical look into the transition

Now that we understand what monolithic and microservices bring to the table, it's time to think about approaching the technical transition. There are different strategies we can adopt, but all of them share the same preparation tasks: identifying boundaries and improving testing.

These preparation tasks are fundamentally important to our success as we deep dive into the transition and cannot be overlooked.

BOUNDARIES

The first thing to figure out before starting the transition is what services are we going to create and how our architecture will look in a completed microservice architecture, how big or how small do we want them to be, and how they will be communicating with each other.

When we think of an application we usually think of a few specific macro-categories of features that have to play nicely together. For example categories like account management, billing management, checkout workflow, search features, and so on. In the early stages of our monolithic applications we are usually able to compartmentalize our code in a good enough way to get the first version of our product out of the door.

As we already know, when building a monolithic application we usually end up with all sorts of functionality in one codebase, most likely talking with the same database. When a rapidly growing product happens to have a highly coupled architecture something not-so-good starts happening as soon as the product and the team grows. Pressure to build new features, deploy them, and iterate over

user's feedback sooner rather than later will transform the codebase negatively and those clean functional boundaries between different features of the product - and the data-model - will start melting together. You will notice this behavior happening when the codebase starts feeling "messy": helper functions all over the place, code that's hard to test, database access done conveniently and improperly. The code becomes extremely entangled and those clean boundaries become more and more blurred.

Keeping clean boundaries is certainly a target of quality that every team should be thriving for, and depending on the maturity of the individual contributors, the review process in place, and the overall engineering culture, it's an achievable goal. But it's also an investment the organization needs to consciously make as an effort to create less technical debt over time, and unfortunately it's an investment that most organizations don't understand or are not willing to make.

Before starting the transition we need to understand the boundaries that make up our system so that we can, in a second step, extract them into services. We start with those boundaries that are more negatively impacted by the monolith, for example those that we deploy, change or scale more often than the others.

TESTING

Transitioning to microservices is effectively a refactoring, therefore all the regular precautions we usually undertake before a "regular" refactoring also apply here, in particular testing.

As we go deeper into the transition we are changing how our system fundamentally works - but we want to be equally sure that afterwards all the functionalities that were once existing in the monolith are still working

in our re-designed architecture. In order to do this it's important that before attempting any change we put in place a solid and reliable suite of integration and regression tests for the monolith.

We do expect some of these tests to fail along the way, and having well tested functionality will help us track down what is not working as expected. I cannot stress enough how important this step is before we attempt to do any change.

Strategies

There are several strategies that we can approach. The first one I like to call the "Ice cream scoop strategy," which allows us to gradually transition our monolithic application to a microservice oriented architecture by scooping out different components within the monolith into separate services. This transition is gradual and there will be times where both the monolith and the microservices exist simultaneously. In this chapter I will focus on this strategy, but first let's explore the other two options.

- + **Pros:** gradually migrating over to microservices with reduced risks without affecting much the uptime and the end user experience.
- **Cons:** it's a gradual process that will take time to fully execute.

The second strategy is for companies that believe their monolith is too big to refactor and prefer to keep it as it is. Instead they choose to only build new features as microservices. Effectively this won't fix the problems with the current codebase but it will fix problems for future expansions of the product. I call this strategy the "Lego Strategy," as we are effectively stacking the monolithic and microservices on top of each other in a hybrid architecture.

- + **Pros:** not having to do much work on the monolith, it's a faster transition.
- **Cons:** the monolith will carry along all of its problems and new APIs will most likely have to be created to support the microservice-oriented features. Could be a strategy to implement to buy some time on the big refactor but ultimately we are adding more tech debt on the monolith.

The third strategy is the "Nuclear Option," rarely adopted. The entire monolithic application is rewritten into microservices all at once, perhaps still supporting the old monolith with hot fixes and patches but building every new feature in the new codebase. Surprisingly I have met a few enterprises who decided to go with this strategy, because they assumed that working on the old monolith was not doable and decided to give up working on it.

- + **Pros:** allows to re-think how things are done, effectively we are rewriting the app from scratch.
- **Cons:** we are rewriting the app from scratch, we may end up with a second system syndrome and the end user will be affected with a stalled monolith until the new architecture is being deployed.

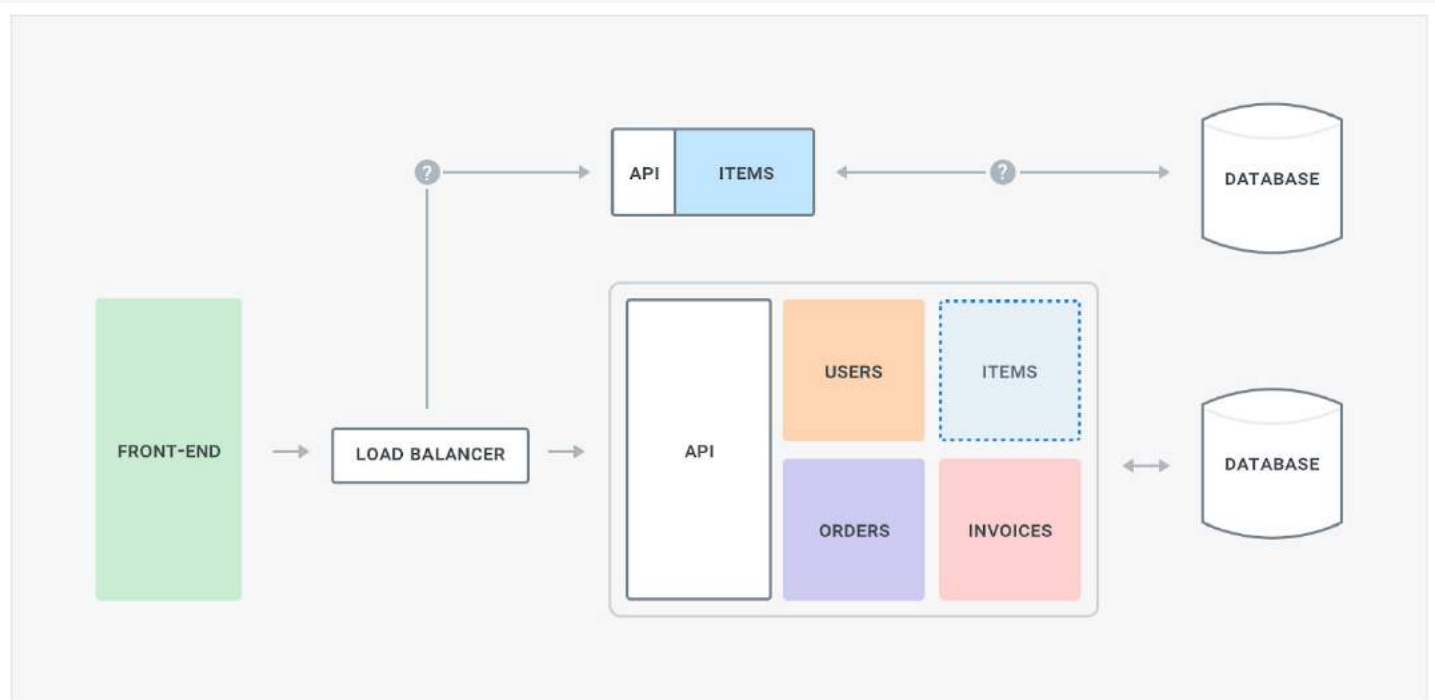
ICE CREAM SCOOP STRATEGY

This strategy assumes that the monolith will be gradually decoupled over time to reduce risks and maximize uptime - we will be scooping out features and services from the monolith into separate microservices that will be communicating with each other in order to provide the same features. This strategy has the advantage of delivering technical and business value along the way, validating the transition as we go deeper and deeper.



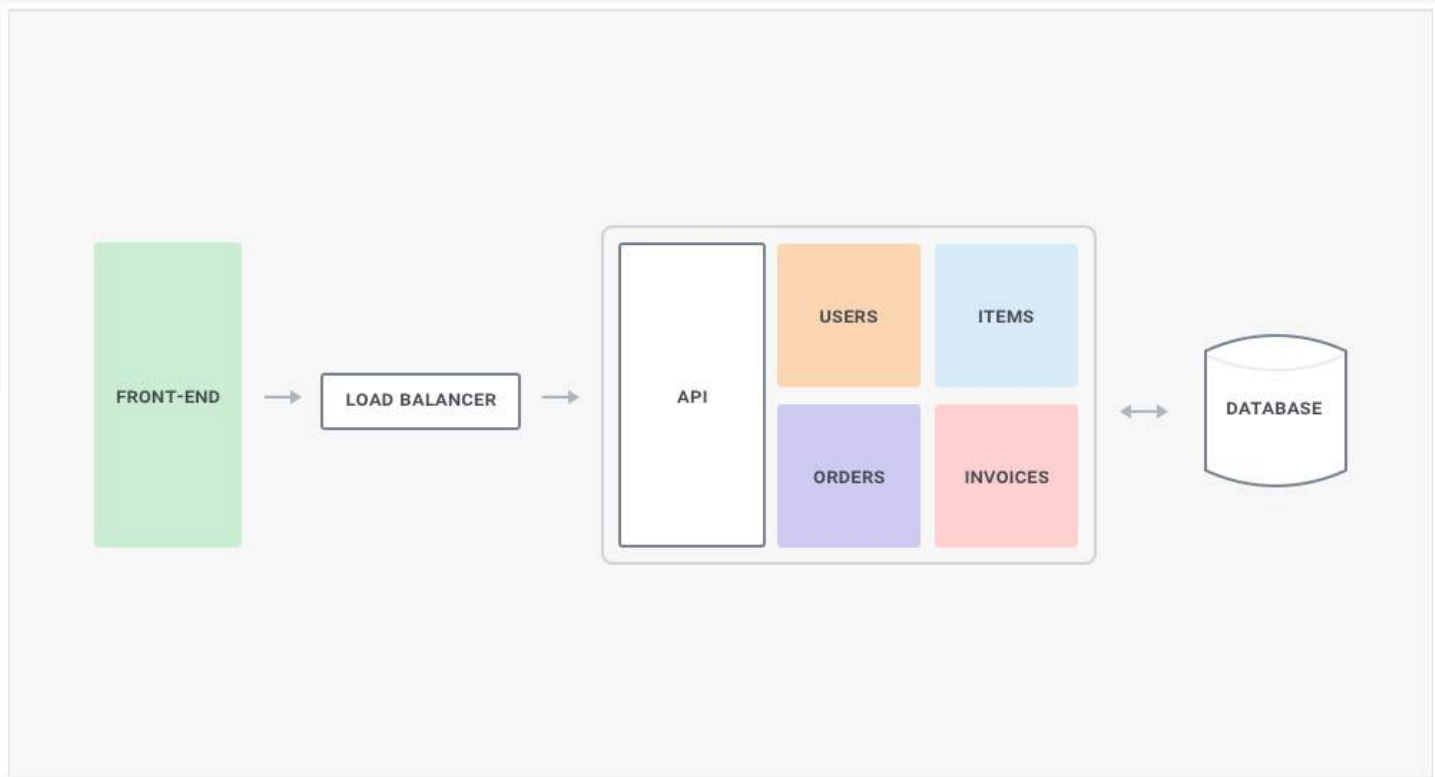
Assuming we have an e-commerce application with a frontend and a monolithic backend, and assuming we have a good test suite in place,

we can identify a few boundaries we could extract: users management, item inventory management, orders and invoicing.



One of the first components we want to separate is the frontend from everything else. The frontend for our application is one of the components that's going to be updated more often - for example with new content or new UI - and we want to start decoupling its lifecycle from the rest of our application. By doing so we are introducing a fundamental component that will shape the next-gen

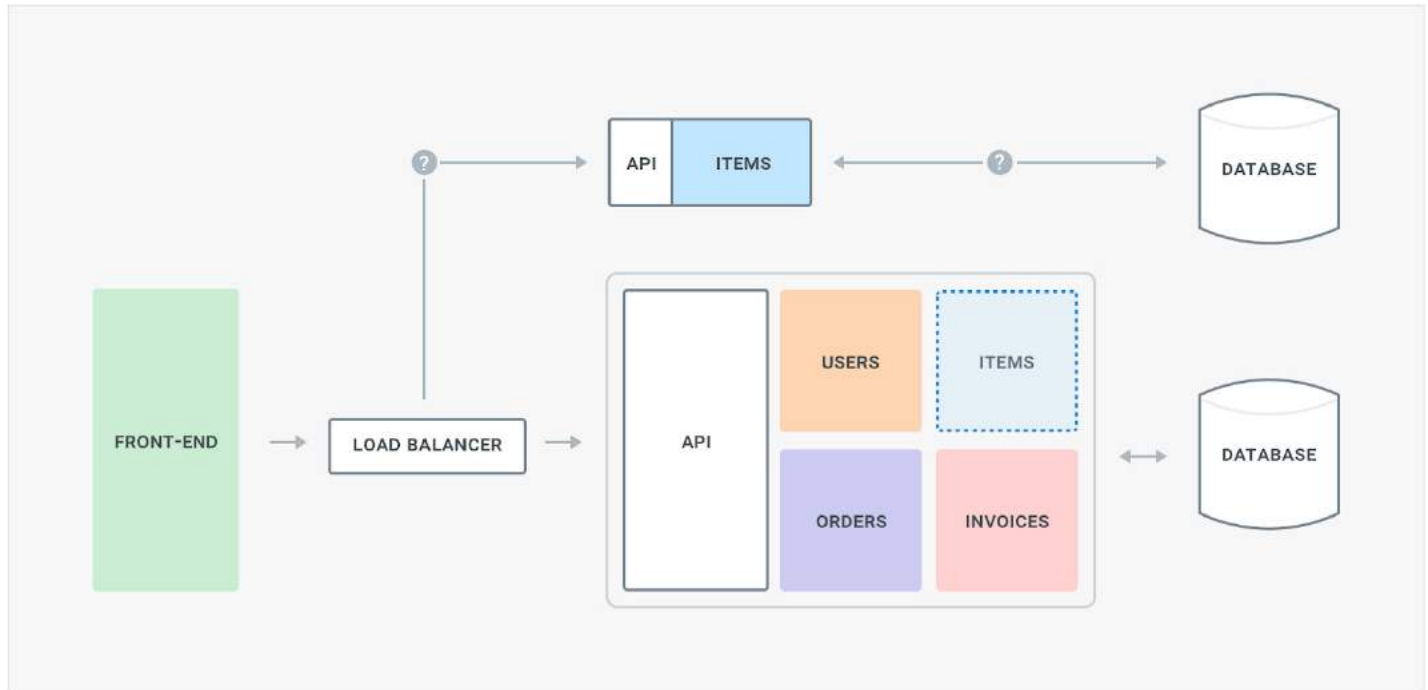
architecture we are building, APIs. In fact, APIs are the backbone of a microservice oriented architecture. We can decide to run the frontend client-side (better choice, more scalable) or still run it server-side (poor choice, less scalable) - regardless of our choice separating the frontend from the backend will introduce the first critical API in our system.



The API is still part of the monolith, but not for much longer. Having an API between the frontend and the backend logic allows us to start decoupling the monolith without disrupting the end users' activity. The API can be built on top of HTTP, RPC, or any other technology, although I would recommend HTTP/1.1 or HTTP/2 (faster) based APIs in order to leverage as much HTTP tooling as we can (for example, HTTP load balancers, caches, etc).

APIs are hard to build, and I will leave it to the reader to research best practices and examples on the topic. In short, breaking changes in an API are extremely painful and an API should be designed since day one to allow for expansibility and versioning.

Moving forward, we can now start extracting other services. For example we could decide that the "Items" management operations (ie, searching or viewing an item, etc) are very intense, and therefore it's our next candidate.



As we start decoupling the first microservice, we quickly realize that we have a few outstanding question marks about handling routing, versioning, code duplication, authentication and authorization, and database access.

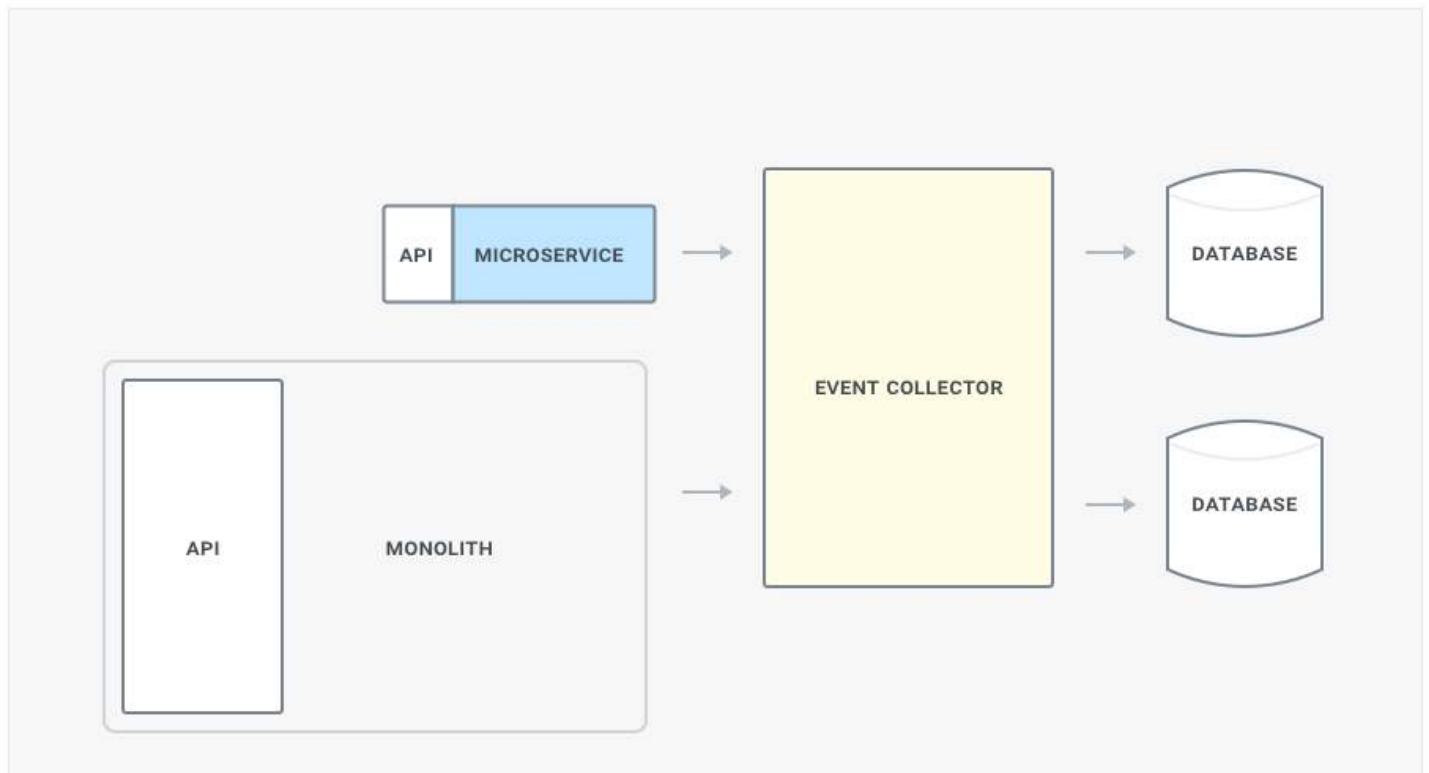
Database

The end goal for the new microservice-oriented architecture is not to rely on one database that every service utilizes. Instead, since we are decoupling the business logic in different services, we also want to decouple database access and have one database for each microservice. Sometimes this is inevitable depending on the use-case the microservice is trying to address: for example, a microservice that handles users and their data will benefit from a relational datastore while a microservice that deals with orders or logs could benefit from high-performance writes of an eventually consistent datastore like Cassandra.

Sometimes some services will indeed use the same underlying database technology, and although it would still be better to have separate database clusters dedicated to each microservice, for low throughput use cases sometimes it's just too convenient to leverage the same datastore nodes but with data stored in different logical databases/keyspaces. For example, you might have two services that both use PostgreSQL, and as long as those microservices are using separate underlying Postgres databases, you could still have one large PSQL cluster that both microservices are talking to. I consider this to be "manageable technical debt" and a good start, but I would still have a strategy in place to start a new dedicated database cluster if one of those microservices becomes too demanding on the db. The cons of this solution is that if a microservice - for whatever reason - impacts the database uptime then the other microservices will also be impacted (since they are talking to the same db nodes), and this is to absolutely avoid down the road since it breaks compartmentalization.

Regardless of your setup we will soon encounter a big problem: consistency of our data. There is going to be a limbo period when the old codebase is still writing and reading to the underlying database, and new database for our microservices will use a separate store for our data. Therefore writes, or reads, made by the monolith won't be visible to the microservice and vice-versa. This is not an easy problem to solve and there are a few options, including:

- 1 Writes from the monolith are also propagated to the microservice database, and vice versa. You will need
- 2 We build an easy-to-use API for the old database that the microservice will use to query data from the old database. You will need to build this API in the monolith and have a temporary synchronization mechanism built into the new microservice.
- 3 We introduce an event collector layer (ie, Kafka) that will take care of propagating writes to both datastores. You will have to build this support in both the monolith and microservice.



Ultimately when the transition is completed, the monolith will disappear and so will this temporary solution.

Routing & Versioning

Every microservice will be accessible by some API of some sort and each microservice will

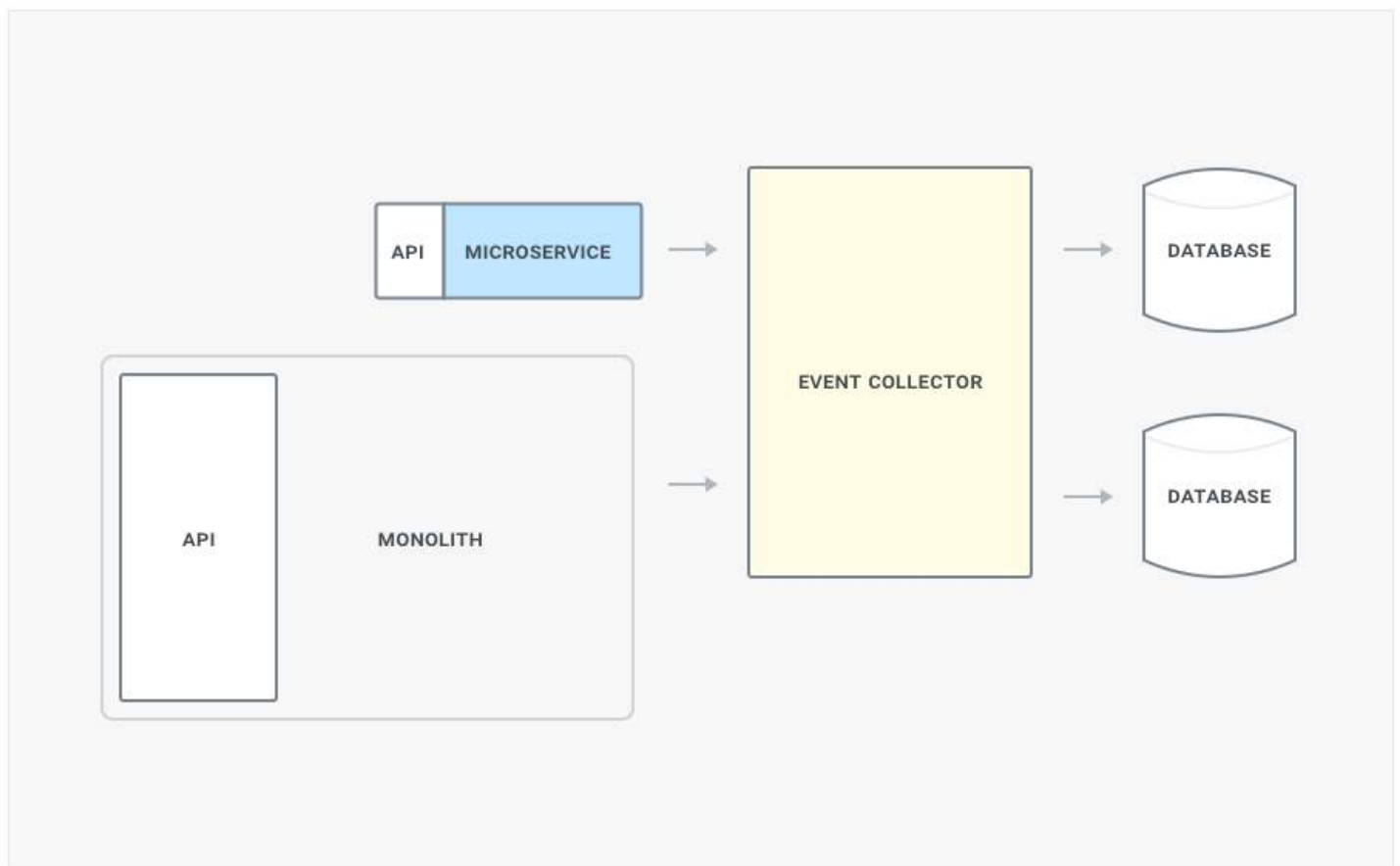
be consuming other services via their API, being totally agnostic of their underlying implementation. APIs are the entrypoint for all of our service-to-service communication. We can therefore make updates to our implementation - including fixing bugs or improving performance - and as long as we don't change the API interface (how requests are being made and their parameters, and the

response payload) other microservices will be able to still work like nothing ever happened.

Every time we make a change we don't want to just route all the traffic to the new version of our service, after all that would be a risky move and if there are any bugs we would impact our application uptime. Instead we want to gradually move traffic over, monitor how the new version behaves, and only when we are confident enough transition the rest of

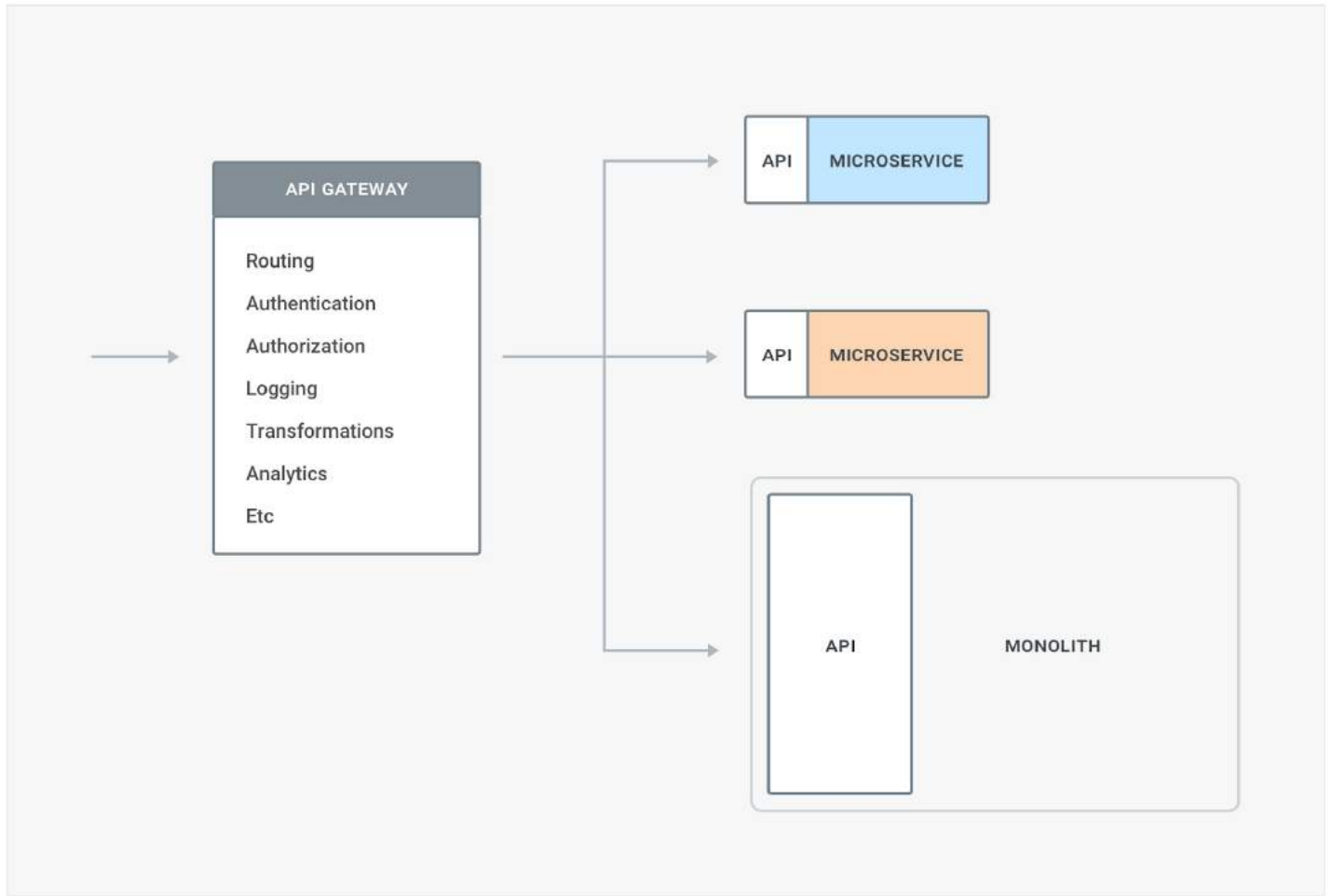
our traffic. This strategy is also called a canary release, and it allows to reduce downtime caused by faulty updates.

This also means we need to have a good strategy in place to route traffic among different versions of our service and monitoring tools in place to determine if any errors have been triggered. This can be done by using proxies, or gateways for north-south traffic, in the following way:



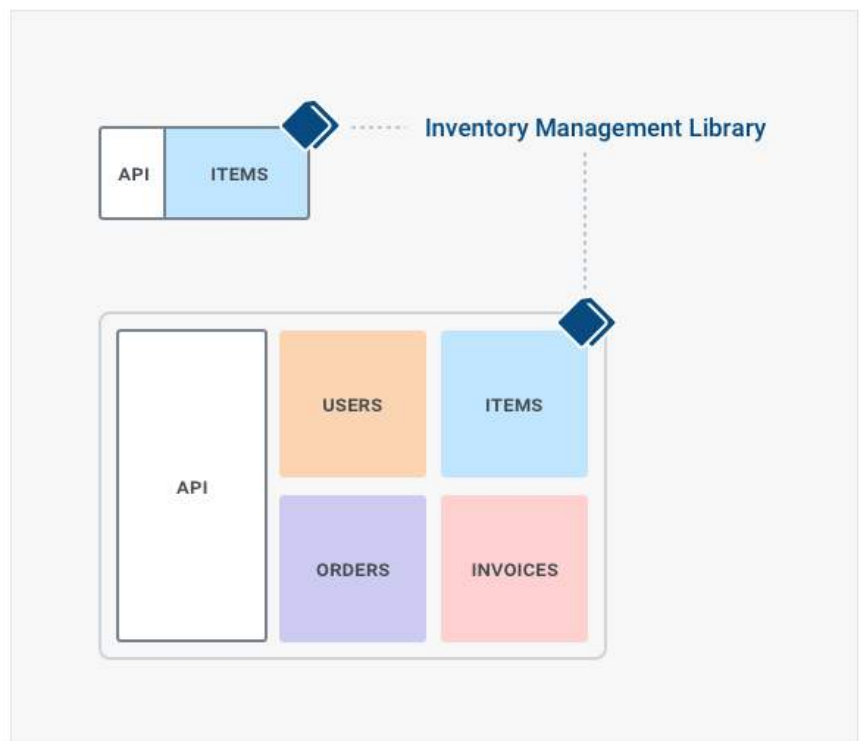
These routing capabilities will be required both when decoupling our monolith and later once you decide to upgrade our microservices to a different version. As you decouple more and more services, you soon realize that they don't live in a vacuum but you need to provide a set of complementary features

like authentication, authorization, logging, perhaps transformations, and this is where API Gateways / Ingress - like Kong - can help in trying to reduce this fragmentation - as well as providing the routing capabilities mentioned above:



Libraries & Security

Our approach needs to be realistic by avoiding to apply too many changes all at the same time. Depending on our codebase we may find it useful to decouple our monolith business logic in third-party libraries that can be then used by both the monolith and the microservice. That also gives us the benefit of reflecting any bug fix or performance enhancement on both codebases while we are in our monolith/microservice limbo.



In general libraries that deal with specific logic which eventually will be implemented by only one microservice do not cause any problems, since any change in the library will require upgrading only one service, but any update to a library used by more microservices simultaneously will cause problems, since any update will have to trigger multiple re-deployments across multiple services - which we would rather avoid since it brings back old memories of the monolith. With that said, there are so many different use-cases on the topic that a one-fit-all answer would be hard to give, but generally speaking whatever roadblock prevents a microservice from being deployed independently or being compartmentalized should be removed in the long run.

Authentication and Authorization were concerns handled internally by the monolith, which can also be implemented within a library, or implemented in a separate layer of our architecture like the API Gateway. Sometimes we will have to re-design how authn and authz are being handled taking into account scalability as we add more and more microservices in our organization. We want to adopt stateless authentication/authorization and perhaps leverage technologies like JWT to achieve our goals.

Security between our microservices should be enforced with mutual TLS to make sure that unauthorized clients within our architecture won't be able to consume them. Logging should be enabled all across the board to observe and monitor how the microservices are behaving.

Now that we have so many moving parts observability becomes a key requirement to detect anomalies, latency and high error rates. Therefore we will need health-checks configured for each one of our services, and circuit breakers that can trip and prevent

cascade failures across our infrastructure if we see too many errors. Since every microservice should be stateless, we can also leverage highly available deployments across multiple availability zones/racks, regions/datacenters with eventual consistency across the entire system. When thinking of microservices it's always a good exercise to assume we are going to deploy the entire system in multiple regions, or even multiple clouds, therefore the decoupling of the monolith needs to address these concerns.

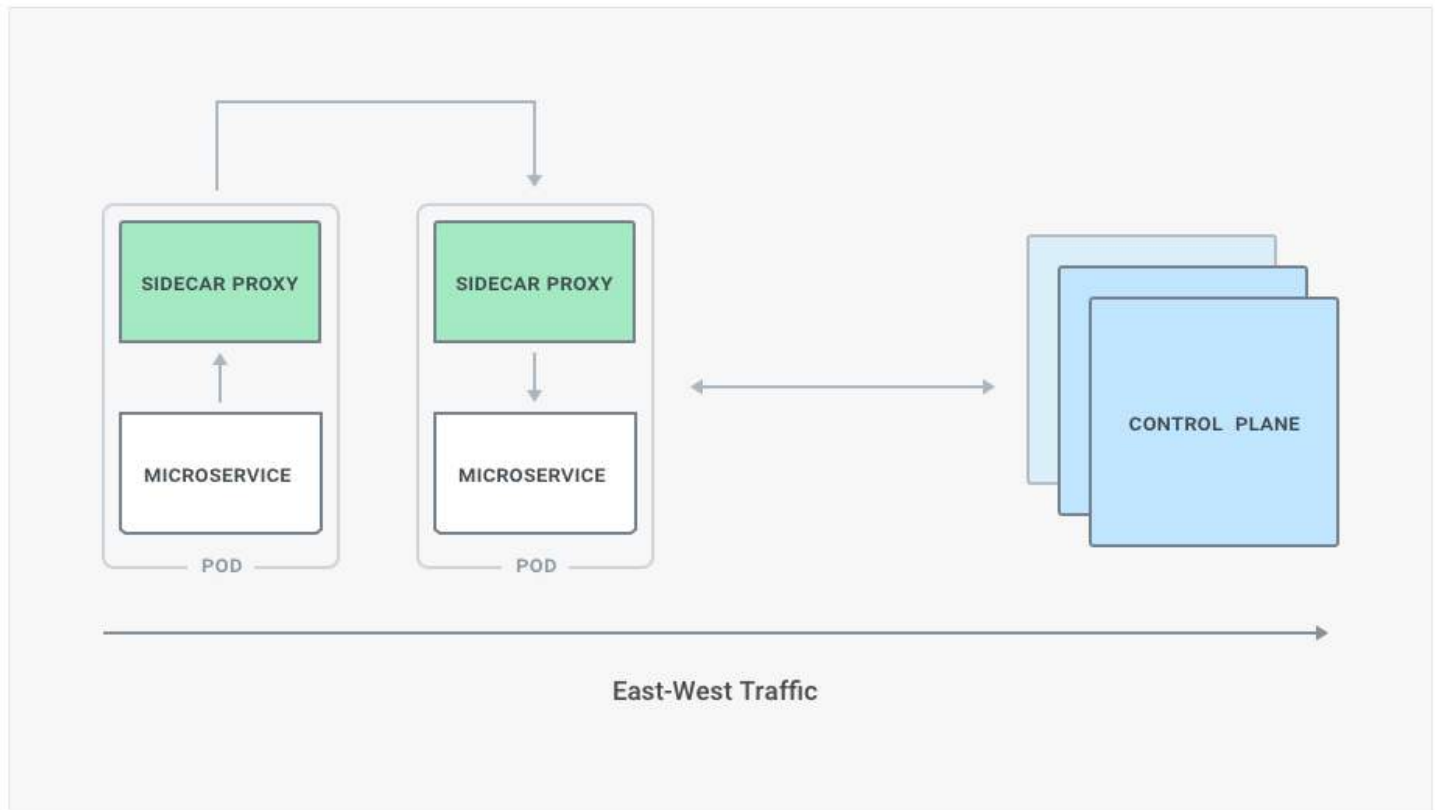
Containers & Service Mesh

Using containers - like Docker - is not technically required although leveraging orchestration tools like Kubernetes, the de-facto container orchestration platform, can make our life a lot easier if we do indeed decide to go through this path. Kubernetes provides, out of the box, many features including facilities to scale up and down our workloads, service discovery and networking capabilities to connect our microservices. In addition to this, Service Mesh is a new emerging architecture design which aims at helping creating microservice-oriented architectures by providing a pattern to perform service-to-service communication delegating features like routing, error handling and observability to a third-party proxy - usually run in a Kubernetes sidecar alongside our microservice processes.

Service Mesh is more commonly used in the context of microservices and the word implies a mesh of services that are all communicating with each other, that is our microservice architecture. The pattern assumes that each instance of our services is not being deployed standalone, but together with a proxy process that sits on the execution path of each request and response. By doing so every time a service wants to consume another service it makes

a request to its local proxy, and the proxy will take it from there and figure out how to communicate with the other microservice. On the receiving end there is another proxy process that accepts the request, and then reverse proxies it to its own underlying microservice process.

The Service Mesh pattern also introduces familiar networking concepts like the control-plane, used to administer our system, and the data-plane, which is primarily used for processing our requests. Both planes communicate together so that configuration can be propagated, and metrics can be collected.



The contact points between each service are the two proxies, so that the services themselves don't have to deal with monitoring, error handling, discovery and so on. The proxies are part of the so called data plane, and they can be configured by another architectural component - the control plane - which is responsible to collect and store user defined configuration and push it to the data plane in a transparent way. The traffic within a service mesh or, more broadly, between microservices in a data-center is called east-west traffic.

Sooner than later these services will have to be consumed by an external client that does not intimately belong to the product whose services belong to, and that communication usually requires an endpoint - or ingress - which will route the request to the most appropriate microservice. A client that would normally go through an ingress point could be another product in the organization, a request coming from another data-center or perhaps an external developer, mobile app or partner.

In the end state of a microservice oriented architecture you will see both service mesh and an ingress playing together to handle their respective - very different - use cases in the best way possible. Service Mesh both as a pattern and as an implementations (like Istio) are still in the early days, but are maturing at a rapid pace and 2018 will see some interesting developments.

Conclusions

Transitioning to microservices is a significant engineering investment with equal significant benefits for applications that reached a certain scale. Therefore Enterprise organizations across every industry are either approaching - or deploying - microservice architectures that can help dealing with the pains of a growing codebase and larger teams. It's both a technical and an organizational transition as we decouple not only our code, but our teams as well.

Approaching such a radical shift cannot be achieved without a long-term plan and preparation tasks that will help us being successful with our transition, including a good testing strategy in place. It's also a feat that's not going to happen overnight, and will most likely include transition implementations that we build to help us along the way, that will have to be removed later on, for example when dealing with legacy database, authentication or authorization functionality.

While personally talking with enterprise architects I keep hearing that they are transitioning to microservices in order to

increase internal re-use and adoption of APIs and services that other teams or departments are creating. While microservices do allow for this consumption mode, if the end goal is to foster cross-departmental collaboration then I would recommend to first consider improving internal North-South communication across those departments first, and then decoupling the monolithic apps powering those APIs as a second step.

Ultimately new patterns and technology are emerging to help promote and build microservice architectures, containers and orchestration platforms being some of them (Docker, Kubernetes), and recently Service Mesh which helps dealing with East-West traffic within the application, and although not a mature solution yet we can expect some interesting developments on its front. API Gateways will help with ingress and so called North-South traffic from both internal and external clients while transitioning away from the monolith, while service mesh can help in managing east-west traffic within the architecture of a specific product. Both service mesh and Ingress will eventually be adopted in the final state of any microservice architecture to deal with their respective use-cases, different but both required at the same time.

Asynchronous patterns should also be explored when an immediate response is not required, since they provide built-in eventual consistency and reduce the number of concerns service-to-service communication brings to the table. Most likely both patterns will be used together to deal with different use-cases, since individually none of them is a one-fits-all solution.



MICROSERVICES FOR STARTUPS // CHAPTER 6

Breaking Up a Monolith: Kong Case Study

MARCO PALLADINO, CTO, [KONG](#)
[@THEFOSK](#)



Kong blew up their monolith. Here's how they did it.

My name is Marco Palladino, co-founder and CTO of [Kong](#), an open-source API Gateway that can be used for internal and external API traffic. In this chapter I am going to write about monolithic applications, microservices architectures, and about the monolithic-to-microservices transition that most organizations, including very large ones, are currently approaching.

We are living in a revolutionary age for software, with massive changes in the way we build, deploy and consume our services. Changes that, as we will learn, are not just technical but organizational as well. New paradigms, patterns and foundations are substantially shaping the industry and we are entering a new era, both technologically and culturally speaking. Open-source software has always been an important contributor

to the industry, but in this day and age it's becoming a major player - like never before - of enterprise adoption to new paradigms and architectures.

Back to the original topic of this chapter - somebody said that decoupling a monolithic application is like going from a whole chicken to chicken nuggets: we are moving away from large codebases to smaller and isolated

services all communicating with each other. And since we have existing traffic and clients currently using our applications, I'd like to add that we need to find a way to keep the chicken alive during the refactoring.

In this chapter, I am going to explore the transition to microservices, with its pros and cons, and deep dive into the technical aspects and common mistakes that I have learned by working daily with developers and enterprise organizations. I will prioritize high priority items that everybody should know before even thinking to transition to microservices, but the topic is ultimately a very large one and I encourage the reader to learn from multiple sources.

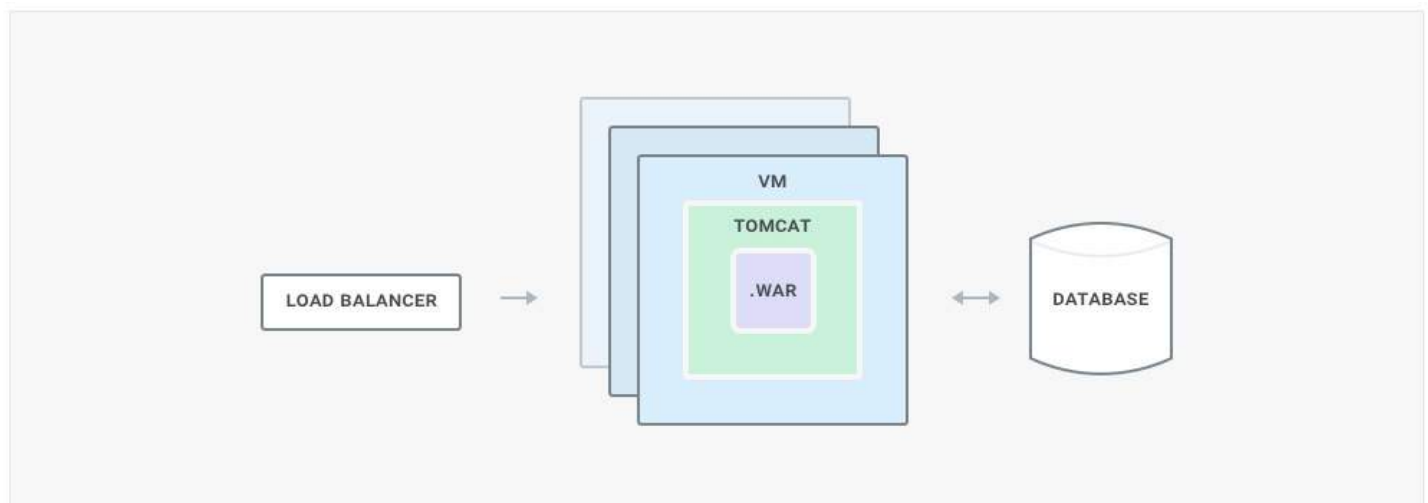
First hand lessons on decoupling the monolith

When in late 2013 I decided to break up our monolithic application into separate services there was little narrative or examples on how to do it. At the time nobody knew about Docker yet - since it had only been around for a few months - and Kubernetes hadn't been released yet, and pretty much most of the tooling available at the time wasn't really a good fit for the sort of

decoupling we were about to approach. Indeed, only a few progressive companies back then have been successful with their monolithic decoupling, and pretty much all of them built their own tooling from scratch to enable the transition, companies like Amazon and Netflix.

At the time Kong, which is today a popular open-source API gateway, didn't exist yet and the company was called by another name: Mashape. Mashape was from 2012 to 2017 the largest API marketplace in the world. If you were an API developer you could search for APIs to consume and if you were an API provider you could publish your services in the marketplace. Mashape was providing the infrastructure to power the marketplace and, in order to authenticate clients and handle billing information, we were processing every single API request to our internal API gateway which, at the time, was enabling about 300,000 developers to consume about 40,000 private and public APIs.

The main platform used to be substantially a monolithic Java codebase hosted on Tomcat and communicating with a single database for pretty much every operation. There was another component, a high performing API Gateway responsible for processing every API request which was built in NGINX + Lua. The gateway technology later in mid-2015 became Kong.



Moving forward I will focus on the monolithic webapp powering the website, excluding the API gateway. Our website was in large parts served via our backend server-side processing, with a few bits left and right written in client-side Javascript consuming ad-hoc APIs to improve the usability on specific pages (like the analytics page) using patterns popular in the early 2010, the time when large JS frameworks like Angular were just about to be released for the first time. We later extended our APIs more and more so that most of the operations could be scripted and automated, but the frontend was largely a mix of client-side API consumption and server-side processing. As every other monolithic application, the backend itself was relying primarily on function calls within the same codebase with almost no outside network call for most of its business logic - excluding database communication.

At the time we were a small team iterating and deploying multiple times per day, and having fewer moving parts allowed us to be extremely agile in the early days. But as the codebase and the team were growing more and more, this approach led to a few problems.

First and foremost deployments became harder. Every small change required a full redeployment of the entire system and lots of team coordination, increasing our overhead and risks. As our codebase was rapidly growing and evolving (we were a startup evolving at a fast pace, after all) what used to be nice and clean boundaries between all the different components of the marketplace (like billing, analytics, catalog management) became more and more entangled over time, making those boundaries more and more blurred. In short, our marketplace fell into the same old problems and tech debt that every monolith experiences when it becomes too large.

When the codebase became increasingly hard to deploy, it made it harder to isolate

failures, harder to onboard new contributors, and we knew something was fundamentally wrong. Slowly but surely these problems also affected the team performance and morale leading to an overall feeling of frustration. Then we finally made the call: we had to move away from the monolith and carve our huge codebase into smaller components. In 2013 there was little prior art on microservice oriented architectures, and as much as I want this story to be glorious, you - the reader - will soon realize that instead I will highlight a series of mistakes which in turn made our system a lot worse and not better. As I talk with teams, leaders and enterprise organizations I am still seeing the same errors happening over and over again even today with the abundance of tooling available, so hopefully this can serve as a warning to some common pitfalls.

Microservices are harder, not easier

When we first decided to transition to microservices Mashape was far from becoming Kong. At the time we were a typical early stage startup with a small team -- too small, as we later found out, to pursue the transition. Although we recognized that the old codebase was effectively slowing us down and something had to be done, transitioning to microservices in an era where microservices were not very well understood, with a small team, rushing through the pushy deadlines (that a small startup needs to constantly meet in order to iterate quickly and make early users happy) was not the most thoughtful decision. In short, we didn't know what we didn't know.

Before starting the transition our plan was simple, or so we thought: we would identify the biggest pain points and boundaries in our monolithic codebase and decouple them in separate services. These services wouldn't necessarily be too small, or too large, but

they would be of the right size required to handle their specific business logic within the boundary they were allocated to, and looking back this was one of the right calls we made early in the process. When it comes to microservices most developers and architects focus too much on the final size, but the reality is that those services will be as big as they need to handle their specific business logic. Too much decoupling, and we end up with too many moving parts. There is always going to be time in the future to decouple them even further as we learn the pain points of building and operating our new architecture.

As we started to approach our transition it became evident that we couldn't just allocate all of our resources to the project, and since our existing business was still running and growing on the monolith we had to split the team in two even smaller teams: one that would maintain the old codebase, while the other one would work on the new codebase. We underestimated resource allocation, and as a side effect this also created some morale friction across the two teams, since maintaining a large monolithic application is not as exciting as working on new technologies. This problem can be more easily dealt with in a larger team. In hindsight, we should have at least considered some sort of team members rotation, and the reason it wasn't done was because we thought it would have slowed us down. And this brings us to the next problem: time.

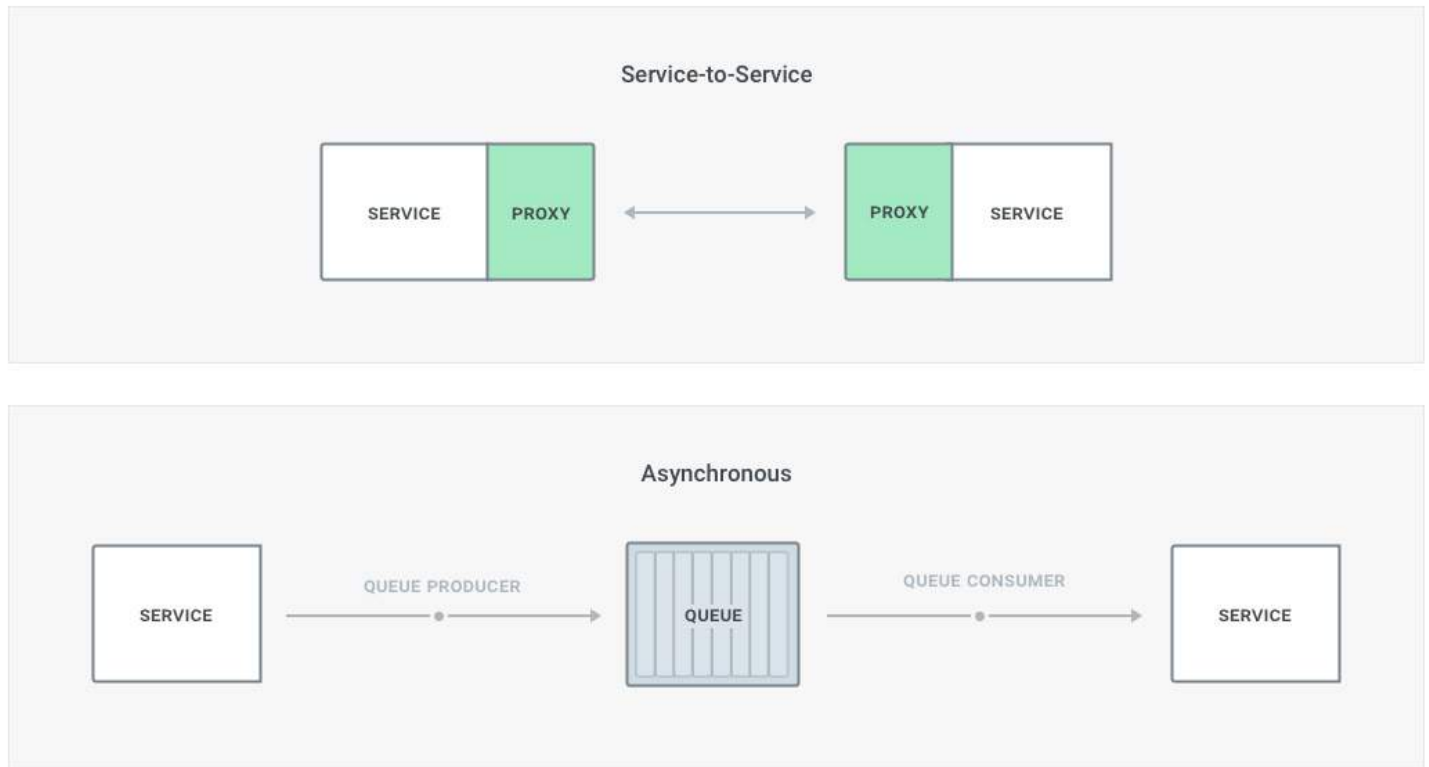
It probably wasn't the right time for Mashape to undertake such a large project. The reason why we decided to reconsider our architecture in the first place was to speed up feature development moving forward. And perhaps with a perfect final microservice oriented architecture we would have been faster, but we didn't realize how painful it would be to break down the monolith and certain areas of the codebase that have been untouched for a long time. As a result of this, we ended up being

slower. Transitioning to microservices requires time, and a lot of it - it can be done in chunks, it can be done with more resources, but there are always going to be bumps along the road that are hard to predict and - as usual in our industry - the excitement of building a better system, and the engineer's ego, can sometime result in the [second-system syndrome](#) and overpower certain concerns. The truth is that transitioning to microservices won't happen overnight, no matter how hard we think about it and how much we plan for the transition, it's just not going to happen quickly. And since the transition can sometimes be longer than the times it takes for a competitive market to move forward, the huge risk is leaving the business behind and damaging the company. Timing the transition is therefore as important as the transition itself.

Two ways for microservice communication

Besides the circumstances described above, one of the biggest lessons learned was understanding when to use service-to-service communication patterns as opposed to what I call asynchronous patterns. And even nowadays - in 2018 - the narrative seems to be exclusively pushing for service-to-service communication, although that's not necessarily the best way to implement very specific use-cases. We learned this the hard way with Mashape when we had to address with critical failure scenarios that could create data inconsistencies across multiple services.

The main difference between the two patterns is that an asynchronous system won't directly communicate with another microservice, but it will instead propagate an event that - asynchronously - another service can intercept and react to it. Usually log collectors like Apache Kafka can be used, but also RabbitMQ or cloud alternatives like - for example - AWS SQS.



This pattern is very useful every time an immediate response is not required, since it basically implements eventual consistency within the architectural pattern since day one and you don't have to worry about it. Therefore it makes a great candidate to propagate those eventually consistent data state changes across every microservice avoiding to create inconsistencies if something goes terribly wrong (for example when one of those services is critically down).

Let's assume for example that we have two microservices, "Orders" and "Invoices," and that every time an order is created an invoice also needs to be created. In a service-to-service pattern the "Orders" service will have to issue a request to "Invoices" every time an order is being provisioned, but if the "Invoices" microservice is completely down and not available eventually that request will timeout and fail. This would be true even if the request was issued by an intermediate proxy (i.e., a

sidecar mesh proxy) as it will still try to make the request over and over again but eventually - if "Invoices" is still down - it will still timeout and fail, leading to data inconsistency in our system.

With an asynchronous pattern, the "Orders" microservice will create an event into a log collector/queue that will be asynchronously processed by the "Invoice" microservice whenever it decides to poll or listen for new events. Therefore even if "Invoices" is currently down for a long period of time, those invoices won't be lost and the data will be eventually consistent across our system as soon as the microservice goes online again (and assuming that the log collector will persist the events).

Using a system like Apache Kafka also makes it easy for the developer to replay a series of events starting from a specific timestamp, in order to reproduce the state of data at any given time either locally or on staging.

Service-to-Service

Microservices and clients directly consume and invoke other microservices.

Ideal for clients that require an immediate response or need to aggregate multiple services together.

Done via HTTP, TCP/UDP, gRPC, etc.

Example: Making a request to retrieve an immediate response of some sort (ie, retrieve list of users).

Asynchronous

Microservices and clients push event into an event collector that's being consumed by other microservices.

Ideal for microservice-to-microservice communication for changing state without requiring an immediate response.

Done via Kafka, RabbitMQ, AWS SQS, etc.

Example: Making a request that doesn't require an immediate response (ie, "orderCreated" event that triggers an invoice creation by another microservice).

Ideally the end state of our microservice-oriented architecture will leverage all of these patterns depending on the use-case, but service-to-service is not the only answer.

There are always going to be times when although a microservice is built with an asynchronous pattern in mind it will have to be consumed directly, so I wouldn't be too surprised if the final state of the service implements both a queue listener and a regular API invocation interface.

At Mashape we eventually learned all the hard lessons, and managed to have a functioning

microservice-oriented architecture down the road, but it definitely wasn't a painless process. Today the industry provides way more tooling and platforms that help the transition than it did in 2013, and it would probably make it easier, although these tools still have rough edges and they are maturing as we speak. I wouldn't recommend starting with a large transition without having first implemented a smaller microservice architecture, perhaps for a new feature or a new small product. Transitioning a large legacy codebase is like opening a can of worms and that prior experience will help making the right calls when the timing is right.



MICROSERVICES FOR STARTUPS // CHAPTER 7

Designing a Successful Microservices Engineering Culture

JAKE LUMETTA, CEO, ButterCMS
[@JAKELUMETTA](#)



You can't build successful microservices without a successful team culture.

Back when I was working with Java developers, I remember there being a source of tension within the camp around who got to work on the newest and meatiest features. Our engineering leadership had decided that we would exclusively use Java to build all new microservices.

There were great reasons for this decision, but — as I will explain later in this chapter — such a restrictive decision come with some repercussions. Note that to the extent possible, communicating the why of a technical decision to the team can go a long way towards creating a culture where people are kept in the loop.

When it comes to organizing and managing a team around microservices, it's always challenging to balance the mood, morale and overall culture of your team. In most cases, the leadership needs to balance the risk of team members using new technology against the needs of the client and the business itself.

This dilemma, and many others like it, has led CTOs to ask themselves questions such as: How much freedom should I give my team when it comes to adopting new technologies? And perhaps even more importantly, how can I manage the overarching culture within my camp?

As in previous chapters, we've turned to the experts for some guidance.

Stability vs flexibility

Let's say you hire a new junior developer. They may be excited about some brand new fresh off the press [JavaScript framework](#). Almost too new.

That framework, while sporting some technical breakthroughs, may not have proven itself in production environments, and it most probably doesn't have great support available. CTOs have the hard choice between okaying that move for the good of morale and excitement within the camp — or declining it to safeguard the needs of the company, to protect the company's bottom line, and to keep the project stable as the deadline approaches.

The right answer depends on a lot of different factors, which also means there is no right answer.

"We give our team and ourselves 100% freedom in considering technology choices. We eventually identified two or three technologies not to use in the end, primarily due to not wanting to complicate our deployment story," said Benjamin Curtis, Co-founder of Honeybadger during [our interview](#).

"In other words, we considered introducing new languages and new approaches into our tech stack when creating our microservices, and we actually did deploy a production

microservice on a different stack at one point. [While we do generally] stick with technologies that we know in order to simplify our ops stack, we periodically revisit that decision to see if potential performance or reliability benefits would be gained by adopting a new technology, but so far we haven't made a change," Curtis continued.

When I spoke with [Stephen Blum, CTO at PubNub](#), he too expressed a similar view in favor of welcoming pretty much any technology that cuts the mustard:

"We're totally open with it. We want to continue to push forward with new open source technologies that are available and we only have a couple constraints with the team that are very fair: must run in container environment and it has to be cost-effective," Blum said.

On the other hand, Darby Frey, Co-founder of Lead Honestly, [takes a more restrictive approach](#) to technology selection.

"At my last company we had a lot of services and a fairly small team, and one of the main things that made it work, especially for the team size that we had, was that every app was the same. Every backend service was a Ruby app," he explained.

Frey told me how this helped simplify the life of his team, as every service has, "the same testing framework, the same database backend, the same background job processing tool, et cetera. Everything was the same."

"That meant that when an engineer would jump around between apps, they're weren't having to learn a new pattern or learn a different language each time. So we're very aware and very strict about keeping that commonality," Frey said.

Having a polyglot architecture can increase the development & maintenance costs. If it's just all the same, you can focus on business value and business features and not have to be super siloed in how your services operate.
—Darby Frey, Lead Honestly

At the same time, Frey is sympathetic to the concept of developers wanting to introduce a new language, admitting that he “loves the idea of trying new things”. However, he feels that the cons outweigh the pros.

“Having a polyglot architecture can increase the development and maintenance costs. If it's just all the same, you can focus on business value and business features and not have to be super siloed in how your services operate. I don't think everybody loves that decision, but at the end of the day when they have to fix something on a weekend or in the middle of the night, they appreciate it,” said Frey.

Give every team member a chance to thrive

Let's revisit my story from the start of this chapter. If you can recall, engineering leaders decided that Java was the best technology to use when building microservices. Java is performant and many of the senior people on the team were well versed with it so that's why we went with Java. However, not everyone on the team had experience with Java.

The problem was, our team split into two camps: the Java guys, and the JavaScript guys. As time went by, new and exciting projects came up, and we'd always reach for Java to

get the job done. Before long, there was some annoyance within the JavaScript camp that crept up.

“Why do the Java guys always get to work on the exciting new projects, while we're left to do the mundane front-end tasks like implementing third party analytics tools? We want a big, exciting project to work on, too!”

Like most rifts, it started out small but grew worse over time.

The lesson I took from that experience was to take your team's expertise and favored technologies into account when choosing a de facto tech stack for your microservices, as well as when it comes to adjusting the level of freedom you give your team to pick and choose their tools.

Sure, you need to have some structure, but if you're too restrictive — or worse yet, blind to the desire of people in your team to innovate with different technologies — you may have a rift of your own to manage.

So, evaluate your team closely, and come up with a plan that empowers everyone in your team. That way, every section of your team can get involved in major projects, without anyone feeling like they're being left on the bench, so to speak.

The question of team organization

The way your team is structured is also going to impact your microservices engineering culture — for better, or worse.

For example, it's common for software engineers to write the code before shipping it off to the operations team, who in turn deploy it to the servers. But when things break (and things *always* break!), an internal conflict occurs.

Because operation engineers don't write the code themselves, they rarely understand problems when they first arise. As a result, they have to get in touch with those who did code it — the software engineers. So, right from the get-go, you've got a middleman relaying a message between the problem and the team that can fix that problem.

To add an extra level of complexity, because software engineers aren't involved with operations, they often can't fully appreciate how their code affects the overall operation of the platform. They only come to know of any issue when operations engineers complain about it. As you can see, this is a relationship that's destined for constant conflict.

One way to attack this problem is by following the lead of Netflix and Amazon, both of which favor decentralized governance. James Lewis and Martin Fowler, two software development thought leaders, laid out their thoughts [via a lengthy blog post](#). According to them, decentralized governance is the way to go when it comes to microservice team organization.

"One of the consequences of centralized governance is the tendency to standardize on single technology platforms. Experience shows

that this approach is constricting - not every problem is a nail and not every solution a hammer," the article reads.

"Perhaps the apogee of decentralized governance is the 'build it, run it' ethos popularized by Amazon. Teams are responsible for all aspects of the software they build including operating the software 24/7," it continues.

Netflix, Lewis and Fowler write, is another company pushing higher levels of responsibility on development teams. They hypothesize that, because they'll be responsible and called upon should anything go wrong later down the line, more care will be taken during the development and testing stages to ensure each microservice is in ship shape.

"These ideas are about as far away from the traditional centralized governance model as it is possible to be," they conclude.

THE CONCEPT OF HIGH-FREEDOM, HIGH-RESPONSIBILITY

During my interview with Sumo Logic CTO Christian Beedgen and Chief Architect Stefan Zier, they expanded on this topic, adding clout to the position that, if you're going to give developers freedom to choose their technology, it has to come with a high level of responsibility attached.

"It's really important that [whoever builds] the software takes full ownership for it. In other words, they not only build software, but they also run the software and remain responsible for the whole lifecycle," they said.

They continued by explaining that a system that resembles a federal government system should be put into place to help keep those freedoms in check by heightening responsibility.

“[You need] a federal culture really. You’ve got to have a system where multiple, independent teams can come together towards the greater goal. That limits the independence of the units to some degree, as they have to agree that there is potentially a federal government of some sort. But within those smaller groups, they can make as many decisions on their own as they like within guidelines established on a higher level,” he said.

Decentralized, federal, or however you wish to frame it, certainly seems to be an attractive approach to structuring microservice teams. It’s an approach that gives each team and each team member the freedom they want — without giving anyone free reign to pull the entire project apart at the seams.

But not everyone agrees.

WHO GETS PAGED ON THE WEEKEND?

You see, a decentralized system implies that each decentralized team takes responsibility for one service, or one set of services. But that too creates a problem: silos.

That’s one reason why Darby Frey, the Co-founder of Lead Honestly, isn’t a proponent of the concept of decentralized governance.

“The pattern of ‘a single team is responsible for a particular service’ is something you see a lot in microservice architectures. We don’t do that, for a couple of reasons. The primary business reason is that we want teams that are responsible not for specific code but for customer-facing features. A team might be responsible for order processing, so that will touch multiple code bases but the end result for the business is that there is one team that owns the whole thing end to end so there are fewer cracks for things to fall through,” Frey explained.

The other main reason, he continued, is that developers can take more ownership of the overall project:

“They can actually think about [the project] holistically,” said Frey.

Nathan Peck, Developer Advocate for Container Services at Amazon Web Services, [explained this problem in more depth](#). In essence, when you separate the software engineers and the operations engineers, you make life harder for your team whenever an issue arises with the code — which is bad news for end users, too.

But here’s what I’m thinking: does decentralization need to lead to separation and siloization?

Peck goes on to explain that his solution lies in DevOps, a model aimed at tightening the feedback loop by bringing these two teams closer together, strengthening team culture and communication in the process. Peck describes this as the, “you build it, you run it” approach.

However, that doesn’t mean teams have to get siloed or distanced away from partaking in certain tasks, as Frey suggests might happen.

“One of the most powerful approaches to decentralized governance is to build a mindset of “DevOps,” Peck wrote. “[With this approach], engineers are involved in all parts of the software pipeline: writing code, building it, deploying the resulting product, and operating and monitoring it in production. The DevOps way contrasts with the older model of separating development teams from operations teams by having development teams ship code ‘over the wall’ to operations teams who were then responsible to run it and maintain it.”

DevOps, as [Armory CTO Isaac Mosquera explained](#), is an agile software development framework and culture that's gaining traction thanks to — well, pretty much everything that Peck said.

Interestingly, Mosquera feels that this approach actually flies in the face of [Conway's Law](#):

Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations.

— M. Conway

"Instead of communication driving software design, now software architecture drives communication. Not only do teams operate and organize differently, but it requires a new set of tooling and process to support this type of architecture, i.e. DevOps," Mosquera explained.

Chris McFadden, VP of Engineering SparkPost, has an interesting example that might be worth following. At SparkPost, you'll find decentralized governance — but you won't find a one-team-per-service culture.

"The team that is developing these microservices started off as one team, but they're now split up into three teams under the same larger group. Each team has some level of responsibility around certain domains and certain expertise, but the ownership of these services is not restricted to any one of these teams," said McFadden.

This approach, McFadden explained, allows for any team to work on anything from new features to bug fixes to production issues relating to any of those services. There's total flexibility, and not a silo in sight.

"It allows [the teams to be] a little more flexible both in terms of new product development as well, just because you're not getting too restricted and that's based on our size as a company and as an engineering team. We really need to retain some flexibility," he continued.

However, size might matter here, as McFadden admitted that if SparkPost was a lot larger, "then it would make more sense to have a single, larger team own one of those microservices."

"[It's] better, I think, to have a little bit more broad responsibility for these services and it gives you a little more flexibility. At least that works for us at this time, where we are as an organization," he said.

On the other hand, Sam Stagg, the VP of Engineering at Pusher, takes Conway's approach — choosing to keep teams working on separate areas of the project:

"We have tried to organize our team to exploit Conway's Law. Thus, we have a small team who builds core shared services for the platform, and services that support specific products are built by the teams who build the products," said Stagg.

A successful microservices engineering culture requires a delicate balancing act

When it comes to technology, freedom — with responsibility — looks to be the most rewarding path. Team members with differing technological preferences will come and go, while new challenges may require you to ditch the technologies that have previously served you well. Software development is constantly in flux, and so you'll need to continually balance the needs of your team are new

devices, technologies and clients emerge.

As for structuring your teams, a decentralized, yet un-siloed approach that leverages DevOps and instills a “you build it, you run it” mentality seems to be popular, although other schools of thought do exist. As usual, you’re going to have to experiment to see what suits your team best.

Here’s a quick recap on how to ensure your team culture meshes well with a microservices architecture:

- **Try to be Sustainable, Yet Flexible:** Balance sustainability without forgetting about flexibility and the need for your team to be innovative when the right opportunity comes along. However, there’s a distinct difference of opinion over how you should achieve that balance.
- **Give Equal Opportunities:** Don’t favor one section of your team over another. If you’re going to impose restrictions, make sure it’s not going to fundamentally alienate team members from the get-go. Think about how your product roadmap is shaping up and forecast how it will be built and who’s going to do the work.
- **Structure Your Team to be Agile, Yet Responsible:** Decentralized governance and agile development is the flavor of the day for a good reason, but don’t forget to install a sense of responsibility within each team.

Thanks to the following individuals for their contributions to this chapter: Ben Curtis, Stephen Blum, Darby Frey, Christian Beedgen, Stefan Zier, Isaac Mosquera, Chris McFadden, Sam Stagg, and Sierra Sterling.

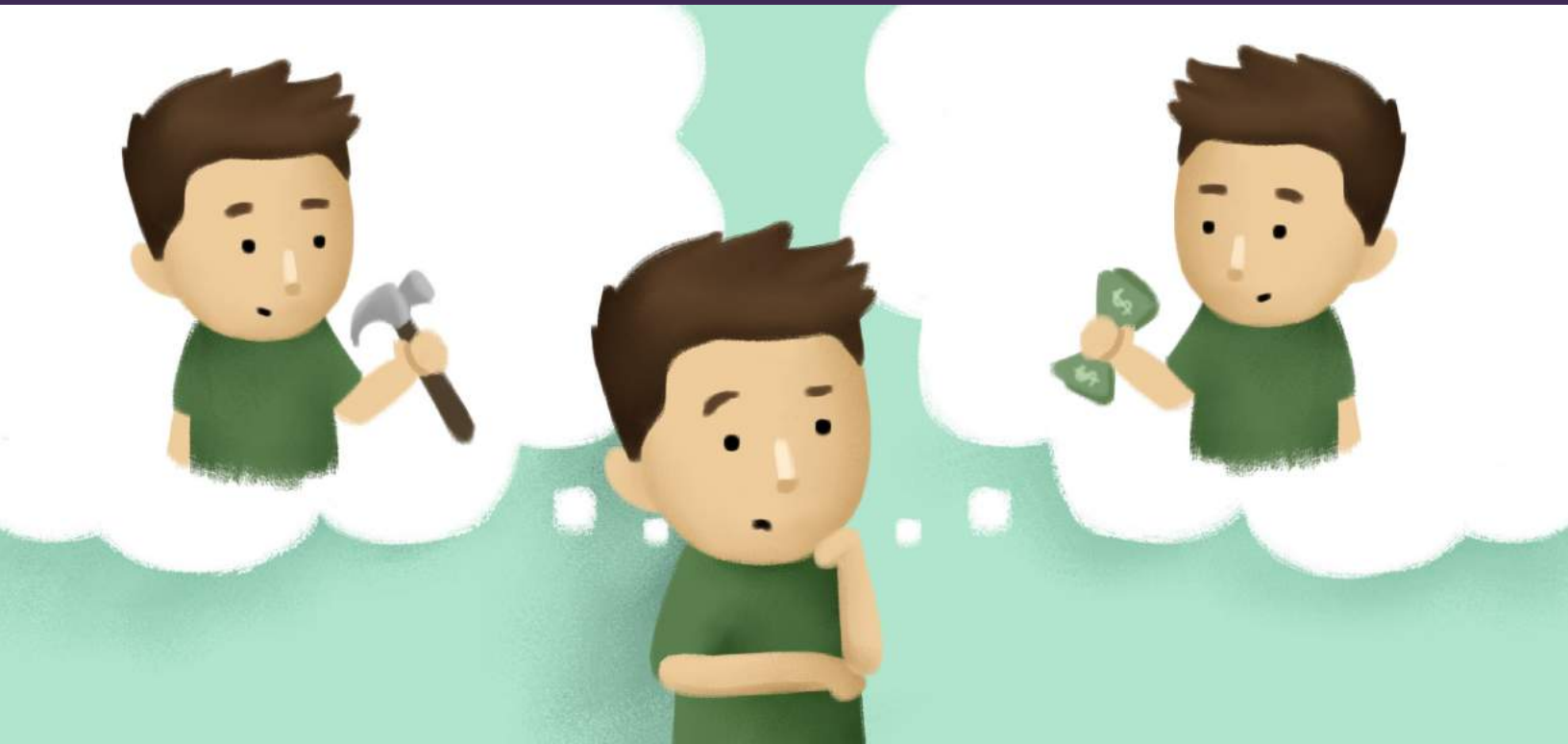


MICROSERVICES FOR STARTUPS // CHAPTER 8

Should You Build or Buy Microservices?

JAKE LUMETTA, CEO, ButterCMS

[@JAKELUMETTA](#)



A look at the microeconomics behind microservices.

As our previous chapters illustrate, there's a lot to consider when it comes to investing in a microservice architecture, from the [culture inside your camp](#) to the [microservice testing strategies](#) to adopt. Another consideration, so fundamental that it's easy to overlook, is not whether you *can* build a microservice but whether you *should* from a business perspective. In short:

SHOULD YOU BUILD THAT MICROSERVICE OR BUY IT?

Because this is a decision that heavily involves all company stakeholders, I've penned this chapter for CTOs who aren't just leading a team of engineers toward their technical vision but for those who also interface with the rest of the business. It's a multi-faceted question, so let's dive in.

Build vs buy: the traditional framework for decision-making

During the initial phase of designing a microservice architecture, you will identify the different components and functionalities

required for the success of your project. Some of these components will have their own individual life cycles and will evolve at different speeds, while others will remain largely untouched. During this process, you'll have to make a choice: build, or buy?

For most developers, the urge to build is overwhelming. I'm as guilty of this as anyone.

We take great pride in our ability to plow through complex obstacles and projects. We can code our way out of anything, we just need a little more time. So when the prospect of buying infrastructure comes up, the typical developer scoffs.

Statements like, "Whatever, I can build that in a couple of weeks, tops!" and, "Why pay that much, when I can build it myself?" are regularly thrown around the office.

And here's the thing, in some cases, those stubborn developers are actually right. They could build a solid solution from scratch. But at other times, they might overestimate their own capabilities, or the time it will take, or the costs it will incur. For example, a talented set of developers might build a microservice during a weekend hackathon, but they might end up spending a month or so finalizing it. It's like a slightly modified Dunning-Kruger effect: you have a talented team, but they may have trouble objectively determining the cost (in time and dollars) of building something from scratch. Even if your team of developers can build the microservice architecture, you have to think about the bigger picture and foresee whether building or buying can help your business deliver value to your customer both efficiently and prudently.

For most developers, the urge to build is overwhelming. I'm as guilty of this as anyone.

Factors to consider when building microservices

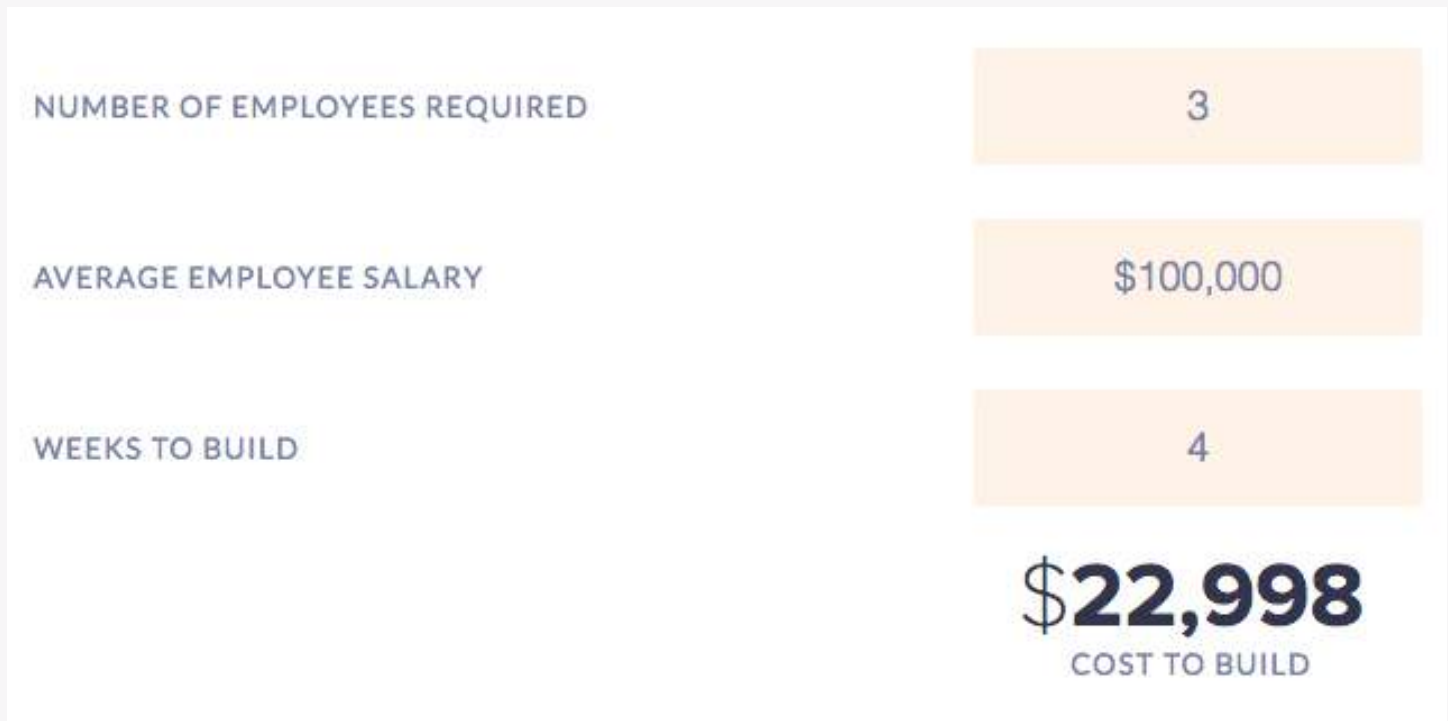
If you do decide to build microservices from scratch, there are costs involved that you'll need to consider before taking the plunge.

DEVELOPMENT COSTS

As a CTO, you know developers aren't cheap and dev costs can quickly add up. You need

to commit developers to create and test the new microservice. You'll also need to factor in the possibility of the development time taking longer than originally anticipated. Whatever effort is estimated, add a 2-3x fudge factor.

For example, let's say your team of 3 engineers, who all make \$100,000 / year, estimates it will take 2 weeks to build a new microservice. In reality before it's production ready, when all said and done, let's say it takes 4 weeks (2x fudge factor). That's going to cost \$23,000 in development time to build that microservice.



Source: <https://baremetrics.com/startup-calculator>

Also consider the issue of [how much technological freedom to give](#) when building new microservices. If you allowed one developer, who was skilled in a bleed edge framework to single handedly build the new service, and then they leave, that will lead to some real costs in headaches in terms of solving that knowledge gap.

MAINTENANCE COSTS

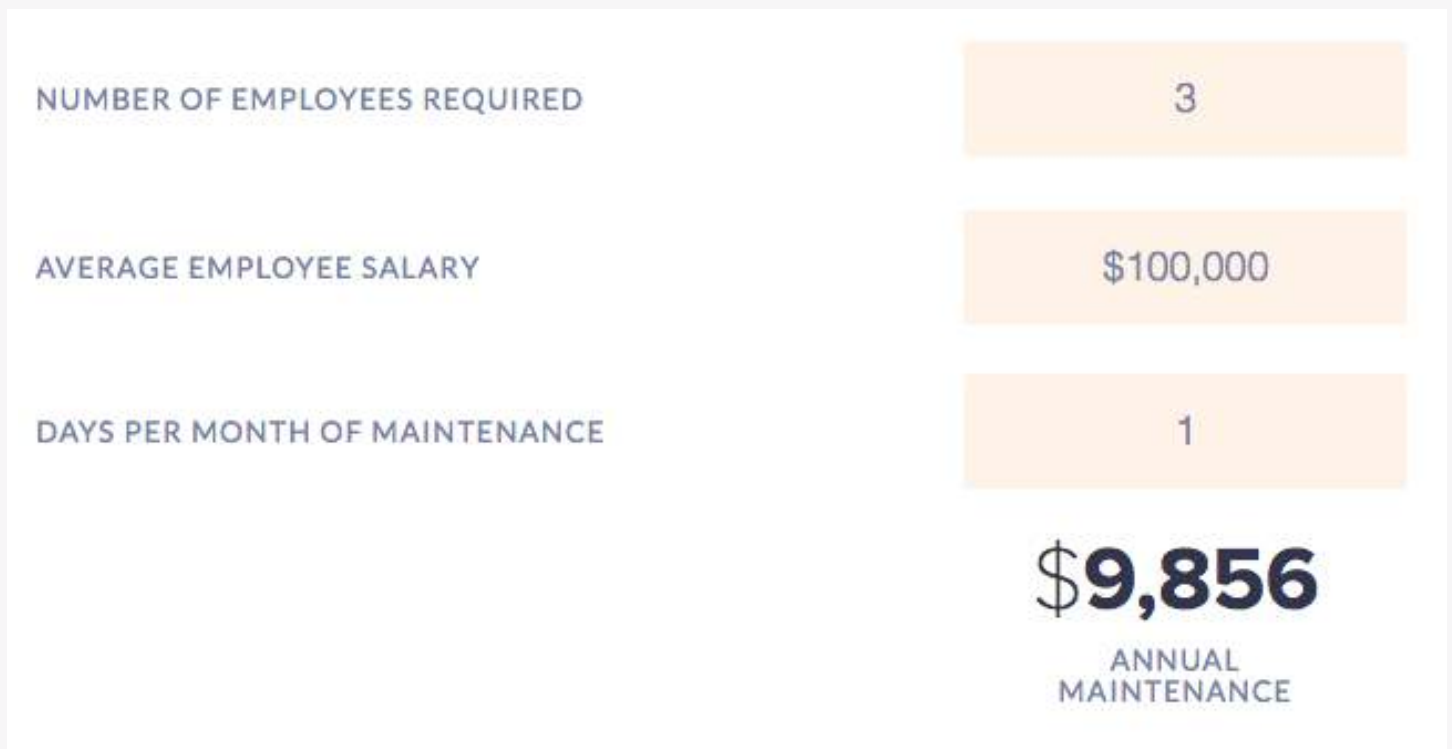
Once your team has created the microservice, now it's your baby. It's your responsibility to look after it and maintain it. As with any software, you'll need to resolve any bugs and issues that arise.

Besides maintaining the system, it is also your team's responsibility to add any necessary

improvements as and when required. If your service executes a marketing function, the marketing team will want to have specific features added onto the service so they are able to carry out their objectives.

But following through on these responsibilities is a case of whether there is enough capacity to do so. According to [Galrath](#), software maintenance makes up 75 percent of the total cost of ownership, which means a large proportion of your developers' time is taken up looking after the service.

Continuing with the example above, let's conservatively assume that your team will spend 1 day per month maintaining (bug fixes) and improving (new features) this microservice. How does that look cost wise?



Source: <https://baremetrics.com/startup-calculator>

SCALING COSTS

As your business grows, your microservices will face more demand and will need to scale up over time. V1 of your microservice may have worked well when you had 100 customers but what happens when you have 1,000 or 10,000 customers? Your microservices will need to scale to meet that demand which means at minimum DevOps time figuring out how to horizontally scale.

SLOWER TIME-TO-MARKET

Putting the dollar costs of having your team build it aside, let's consider the reputational cost of building instead of buying.

First off, building your microservice architecture from scratch entails a slower time to market. Slower time to market means a later launch

date of your product or service — which in turn postpones any revenue coming in.

Worse yet, your brand itself may suffer as competitors reach your target market more swiftly, leaving you at a disadvantage in the market — when you eventually get there. If it isn't part of your core functionality, why are you spending your time doing it when you can just buy it?

REPUTATIONAL COSTS

Let's be frank, building out new capability — particularly when you're trying to do it on a budget and within a strict time frame — can be difficult. Worse, despite how hard your team works on it, if in the end it functions like a product of a rushed timeline, it's a bad look for your company. If it doesn't work or isn't polished, customers don't care how many late nights were spent building it.

Factors to consider when buying microservices

Buying helps you leap-frog most, if not all, of the negative factors associated with building microservices. Yet, there's still a lot to consider before deciding to buy capability over building:

VENDOR REPUTATION

There are many tools and products that are readily available for you to plug straight into your stack. But in an ideal world, you want something that has been tried-and-tested and has an established reputation.

A good way to confirm the reliability of a service is via social proof. Look out for testimonials, reviews, ratings and recommendations. The more positive and authentic the social proofing, then you know it is a reliable service.

SUPPORT

If anything were to go wrong with the service, you want to make sure you can readily access support if anything were to happen. It's also wise to see if the support is readily accessible. Check to see if the service provides supports through web chat, email or phone. Ideally, you want something that has 24/7 emergency support.

INTEGRATIONS

You want a service that can easily plug into your stack. Check for compatibility when shopping around for microservices. You should also look for something that will be, to the extent possible, be future proof as the technology landscape evolves.

When it makes sense to build

There are two scenarios when opting to build is the best solution.

The first is if your microservices make up the core functionality of your organization and are the key differentiator for your business. When it comes to technology, keeping the core of your product or service proprietary is always preferred.

Secondly, it makes more sense to build if you are simply unable to find a product in the market that suits your needs. That's rare in today's market, but not impossible.

When it makes sense to buy

If the functionality you need is not part of your product's core, and is therefore not a key differentiator in the market — like authentication or content management software for example — it makes a whole lot more sense to buy. That's especially true if you come across a high-quality vendor that can easily integrate with your existing stack.

It's also wiser to buy if you find a quality third-party service or API that can add functionality to your product.

Finally, perfect is the enemy of good enough. Not every dimension of your company needs perfectly bespoke functionality — sometimes, it's better to get something off the shelf that does a good job, rather than spending time and money on trying to build something "perfect."

The Market is Revealing

There's a booming market out there for APIs and SaaS products right now, because

they work. They plug the gap that stubborn developers try to fill with good intentions, talent — and unfortunately for the company — a whole lot of time and money.

But don't take my word for it. The 2018 [MarTech Supergraphic](#) documents 6,829 marketing technology solutions, an increase of 27 percent from the previous year. It's clear that there's a growing demand from companies looking to buy rather than build — especially when it comes to technologies that aren't key differentiators.

A CMS is the perfect example of a microservice (or set of microservices) that is not a differentiator for most businesses. It's an internal tool used to help you create, manage and publish content, so although it impacts the customer experience, a CMS is never what sets you apart from your competition.

Thus, a CMS is the right kind of technology to buy, rather than build, especially if you can find a CMS that serves a purpose today, as well as tomorrow. ButterCMS, for example, is a headless CMS that enables brands to use [content as a microservice](#) by decoupling content from the front-end delivery layer. Moreover, since ButterCMS is SaaS and leverages CDNs and APIs so that as your customers and traffic grow, all of the content scaling is handled for you.

For other examples we can look to companies featured in this book including [Algolia](#) for search, [Scalyr](#) and [Sumo Logic](#) for log management, [Pusher](#) and [PubNub](#) for real-time data streaming, etc.

Reinventing the Wheel is Never Cost Effective

To recap, let's quickly revisit the two scenarios where building is better, and the three

scenarios where buying microservices makes more sense:

BUILD MICROSERVICES WHEN

- 1 The microservices make up the core functionality of your product, service, or organization, and are therefore a key differentiator for your business
- 2 You are unable to find a product in the market that meets your needs

BUY MICROSERVICES WHEN

- 1 The functionality you need is not part of your product's core, and is therefore not a key differentiator in the market
- 2 You find a quality third-party service or API that can add functionality and depth to your product or service
- 3 You want to escape the chase for perfection, and need a solution that gets the job done today, rather than an abstract "perfect" idea that will take months to build and maintain

Although technical ability plays a role in this decision, it really comes down to cost and efficiency and letting your team focus on what your company does best and letting others handle the stuff you aren't excited about.

Going back to our earlier build economics example, let's assume you found a SaaS service that costs \$200/month that you're considering buying as an alternative to building it yourself. Admittedly, if you're a startup, \$200/month is enough to pause and thinking about as it seems like a lot of money. Let's see how it compares to building:

IF YOU DON'T BUILD IT	
ANNUAL COST OF PAYING FOR AN ALTERNATIVE SERVICE	\$2,400
IF YOU DO BUILD IT	
NUMBER OF EMPLOYEES REQUIRED	3
AVERAGE EMPLOYEE SALARY	\$100,000
WEEKS TO BUILD	4
DAYS PER MONTH OF MAINTENANCE	1

Don't build! It's not worth it!

This project is not going to work out.

\$22,998	\$9,856	-\$7,456	Never
COST TO BUILD	ANNUAL MAINTENANCE	SAVED ANNUALLY	YEARS TO PROFIT

Source: <https://baremetrics.com/startup-calculator>

When you boil it down, and look at the real costs in the macro, the question of build versus buy can be straightforward. As CTOs, we need to always take a strategic view of what's best for the business, even if deep down our inner software developer (and/or our team) is chomping at the bit to prove we can build it.

Thanks to the following individuals for their contributions to this chapter: Ben Curtis, Stephen Blum, Darby Frey, Christian Beedgen, Stefan Zier, Isaac Mosquera, Chris McFadden, Sam Stagg, and Sierra Sterling.