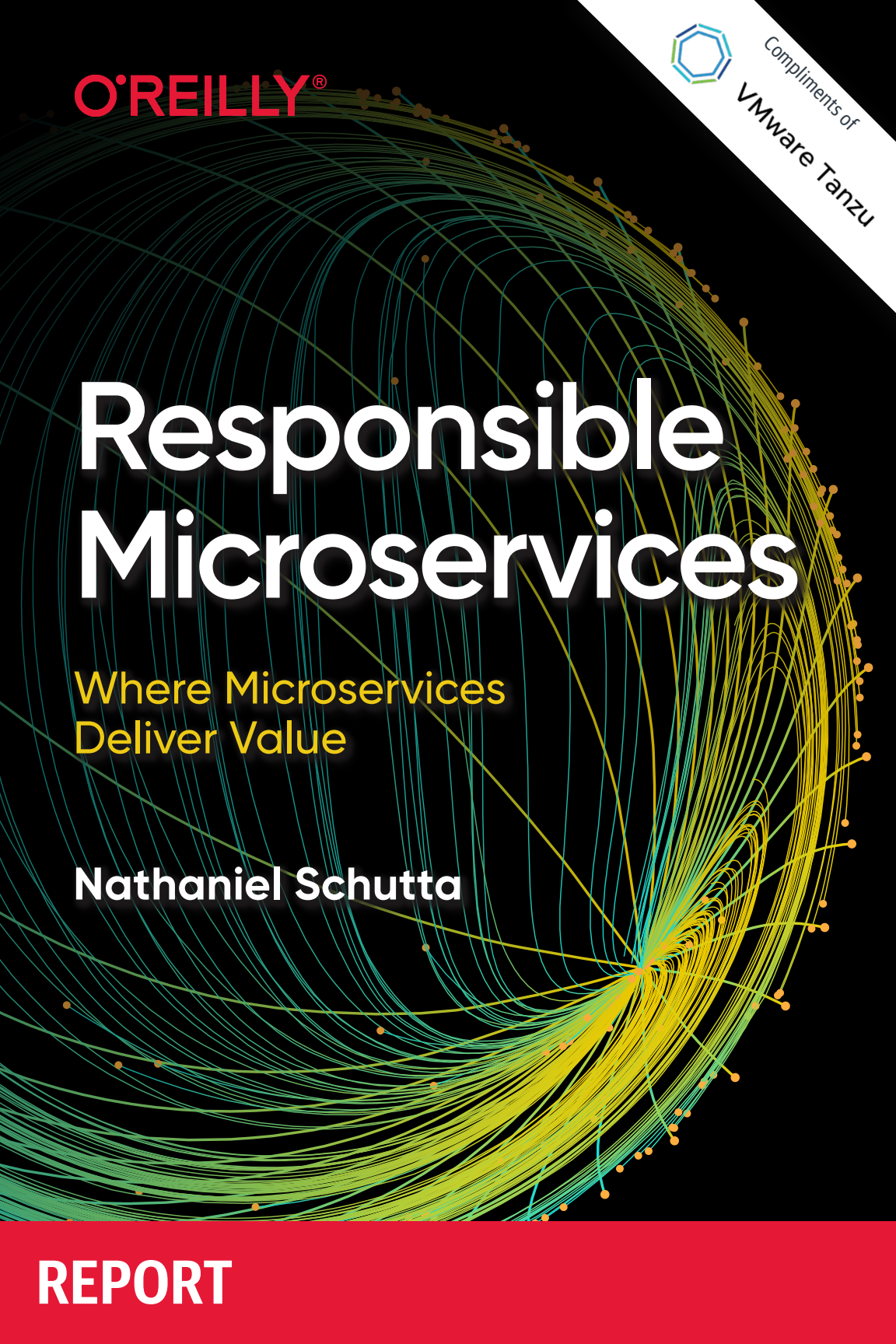
O'REILLY®
Compliments of
VMware Tanzu
Responsible Microservices
Where Microservices Deliver Value
Nathaniel Schutta
REPORT

VMware
Pivotal Labs

Build software a smarter way

Jumpstart app development in an iterative, results-driven way. We help you deliver great apps with proven practices and simple tools. You'll have working software in days, thanks to an approach that starts small and scales fast. Build new apps your customers love and update the ones they already rely on.

Modernize your existing apps
Build innovative new products
Collaborate in a culture of continuous learning

https://tanzu.vmware.com/labs

Responsible Microservices

Where Microservices Deliver Value

Nathaniel Schutta

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Responsible Microservices
by Nathaniel Schutta

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Melissa Duffield
Development Editor: Melissa Potter
Production Editor: Kate Galloway
Copyeditor: Piper Editorial, LLC
Proofreader: Piper Editorial, LLC
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: O'Reilly Media, Inc.

August 2020: First Edition

Revision History for the First Edition
2020-08-14: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492085287* for release details.

978-1-492-08526-3

[LSI]

Table of Contents

CHAPTER 1
The Microservice Revolution

These days, you can't swing a dry-erase marker without hitting someone talking about *microservices*. Developers are studying Eric Evans's prescient book *Domain-Driven Design* (Addison-Wesley). Teams are refactoring monolithic apps, looking for bounded contexts and defining a ubiquitous language. And while there have been countless books, videos, and talks to help you convert to microservices, few have spent any appreciable time asking if a given application *should be* a microservice.

There are many good reasons to use a microservices architecture, but there are no free lunches. The positives of microservices come with added complexity. Teams should happily take on that complexity, provided the application in question benefits.

This report will give you a set of principles you can use to help focus your efforts and avoid wasting your time. As you read through the following pages, ask if your application benefits from a given principle. If you answer "yes" for one or more of the following principles, the feature is a good candidate to be a microservice. If you answer "no" for every principle, you are likely introducing *accidental complexity* into your system.[1] But why are so many companies adopting microservices in the first place? What even *is* a microservice?

1 For more on this concept, see *The Mythical Man Month* by Frederick P. Brooks Jr. (Addison-Wesley).

How Did We Get Here?

Odds are you've noticed a major shift in how your organization approaches infrastructure. Servers were once homegrown—a bespoke artisanal approach. And while you may enjoy the idiosyncratic when it comes to your morning coffee or your favorite food truck, unnecessary variables in your infrastructure lead to sleepless nights. During this "Paleolithic" era of software, servers were a very expensive resource, forcing developers to deploy as many apps to the same hardware as possible. Doing so may have placated the accountants, but it introduced its own set of problems.

With shared resources, one application's bug could impact every application on a given box. Upgrades to common libraries were constrained by the slowest-moving system in the environment, making currency projects a frustrating series of freezes and testing. Organizations often kicked the can down the street rather than deal with vital (but not flashy) currency projects. Afraid of breaking anything, many companies poured proverbial concrete over their infrastructure, allowing fear to lead them down a dark path. Who knew there was such a thing as #YodaOps?

> Fear is the path to the dark side. Fear leads to anger. Anger leads to hate. Hate leads to suffering.
>
> —Yoda, *Star Wars Episode I: The Phantom Menace*

Delivering code to production was its own source of frustration as well. As an application moved from dev to test and beyond, things that worked in one region were just as likely to stop working in a different region. You could spend days pounding your head against the wall trying to determine what, exactly, was amiss, wasting countless hours that could have been better spent delivering features and functionality.

What Would You Say Is Different Here?

Like nearly every developer, I've fallen victim to the "works here but not there" conundrum multiple times in my career, but one case in particular stands out. Our project was cruising along until we got to the customer test, only to be foiled. After a couple of weeks of meetings, emails, and escalations, we finally had an answer. The difference? The order in which the patches were applied. It was at this point that I seriously wondered what life choices had led me to this place.

The software industry doesn't stand still; in fact, it seems to be in constant flux.[2] Infrastructure is a different game today, and servers are commodities. Rather than spend countless hours troubleshooting a bad server, it is faster to just destroy the instance and spin up a fresh one. With public cloud providers, containers, and app platforms, you now bundle your application with everything it needs and move *that* abstraction from server to server. In truth, you probably aren't *moving* anything, you're just updating a routing table.

With these higher-level abstractions, if it works in dev, it will work in test because you are working with the *exact* same thing, eliminating an entire class of bugs from the procedure. It also liberates development teams—they are no longer subject to the tyranny of the slowest-moving application. If *your* application needs a spiffy new library version, go ahead, you aren't affecting anyone else! You can focus your attention on solving problems for your customers, not undifferentiated heavy lifting.

Your teams can deliver in days or weeks instead of months or years, allowing you to be far more responsive to business changes. You can run *A/B tests* and perform hypothesis-driven development instead of hazarding guesses and arguing in the project room. Disruption affects every industry, and you can no longer afford to rely on "We've always done it that way." You must evolve just as one large bank did (described in Figure 1-1). If an organization in a heavily regulated field such as banking can adapt, so can you.

2 See *Thinking Architecturally*, another report from O'Reilly, for more on this concept.

Figure 1-1. You can move to thousands of deploys a month

None of these benefits magically happen; they are the culmination of cloud environments, cloud native architectures, DevOps, and the cultural shift inherent in any transformative technology. The transition takes time, but the results speak for themselves, allowing you to deliver business critical software consistently and repeatedly.

Microservices are ultimately a reaction to plodding monoliths and heavyweight services, as well as modern cloud environments. Monoliths suffer from a lengthy list of problems, starting with long ramp-up times for new developers, all the way to build times measured in phases of the moon. With years (or decades) of technical debt, modularity breaks down over time, making it very difficult to refactor and add vital new features. Scaling typically means adding capacity for the entire application, not just the pieces that needed it, leading to single-digit resource utilization. Out of this frustration was born the microservice.

What Is a Microservice?

There are nearly as many definitions of a microservice as there are developers touting them as miracle cures. Before delving further, the key definition is the one inside the walls of *your* organization. Whether it adheres to the Platonic ideal form of a microservice isn't nearly as important as getting everyone on the same page. There is a reason why a glossary is often one of the most important artifacts in any project room.

What's in a Name?

If you've already debated tabs versus spaces, consider touching off a discussion around the definition of a microservice. Consider removing sharp objects—it may devolve rapidly. Microservices really are in the eye of the beholder!

Ultimately, microservices are a reaction to monoliths and heavyweight *service oriented architectures* (SOA), as well as the capabilities of cloud environments.[3] The issues with poorly structured monolithic architectures are legion, from low developer productivity caused by massive codebases to the inability to target compute resources to the bits that need more performance, there are no shortage of headaches. Software is not immune to the second law of thermodynamics; over time, the modularity of the monolith breaks down and it takes longer and longer to add new features and functionality.

Some are partial to defining a microservice as any service built and maintained by a two-pizza team. Personally, I am a fan of defining them as something that can be rewritten in two weeks or less, since that reminds us that microservices should be, well, small. Of course, there is no stock answer to the question of how microservices should be defined—it depends on the volatility of the services in question. While I support two-pizza teams, that definition won't help you determine just how many services said team can support. If the microservices are stable, a two-pizza team might be able to support ten or twenty of them. However, if the services are

3 According to some people, microservices are SOA done right.

constantly changing, the exact same team might struggle with more than five!

Rather than debate designations, think in terms of characteristics. Microservices are suites of small, focused services that embody the Unix ethos of small, focused tools that do one thing and do it well.[4] Microservices should be independently deployable, independently scalable, and free to evolve at different rates. Developers are free to choose the best technology to build services around business capabilities. In a nutshell, microservices are an example of what I refer to as the zeroth law of computer science—*high cohesion, low coupling*—applied to services.

Microservices Are a Tool

At the end of the day, microservices are a tool and it is up to *you* to properly apply them. It is just another approach. An architectural style. A pattern. It is not Mjölnir, and it will not solve every problem you've ever had on a given project.[5] If you've ever been to a home improvement store, you might have noticed there is an entire aisle full of hammers. Some are smaller, some are larger; some have smooth faces, some knurled. Some include a handy hook to help you pull out the nail you inevitably bent, while others will help you tear down a wall with minimal fuss.

The expert knows when to pick up which hammer, while the novice often falls in love with the first hammer they ever used. If you are demolishing a shed, a sledgehammer is your friend. While it will help you get rid of that eyesore, using it to put up some beautiful maple trim will probably result in a trip to urgent care. You face a similar choice with your applications. Are microservices useful? Absolutely. Are they right for every situation? Of course not.

4 See Eric S. Raymond's "Basics of the Unix Philosophy" for more.

5 In case your Norse mythology is a little rusty, *Mjölnir* is Thor's hammer. You may recall that it was destroyed by Hela (in the Marvel Cinematic Universe, at least).

Microservices really do offer some impressive benefits. But they come at a price. Don't pay the complexity tax unless you get something in return. In other words, no, not everything should be a microservice! Use them where they make sense. Use them where they add value. If you need one (or more) of the principles that follow, go forth and prosper! If not…well, there's always *serverless*.

CHAPTER 2

Multiple Rates of Change

Do parts of your system need to evolve at different speeds, or in different directions? In any system, some modules are hardly touched while others seem to change every iteration. Separating them into microservices can be useful, allowing each component to have independent life cycles.

Parts of Your System Evolve at Different Rates

It should be obvious to anyone involved in a software project that not every part of your system evolves at the same rate. Some components are changed constantly while others haven't been modified in weeks or months, even years. If aspects of your system evolve at different speeds, you might need microservices. To illustrate, let's pick an example—say, a monolithic online retail app, as described in Figure 2-1.

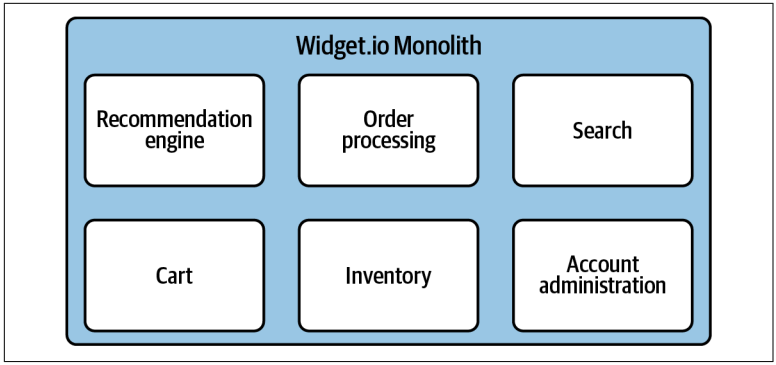

Figure 2-1. For your consideration…the Widget.io Monolith!

Odds are, the cart module probably hasn't changed much and the inventory system is really stable.[1] Meanwhile, your product owner constantly wants changes to the recommendation engine, and no one has ever said "Our search is too good." In the monolith, everything has to move at the same rate, which is part of the rationale behind quarterly release cycles.

Today we have options. Splitting those two modules—recommendation engine and search—into microservices would allow the respective pieces to iterate at a faster pace. This approach will help you quickly deliver business value. Separating the things that change more frequently results in something like Figure 2-2.

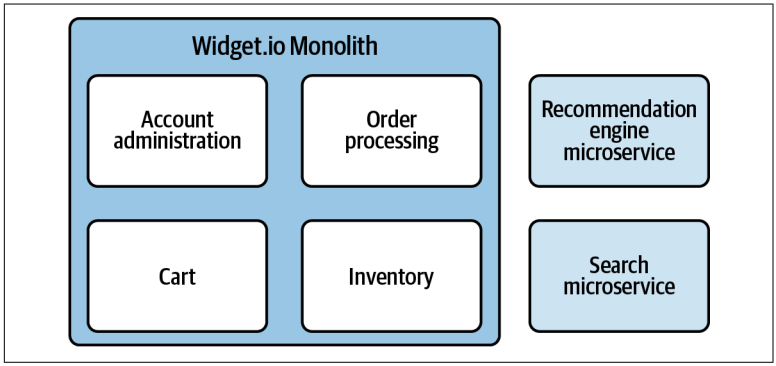

Figure 2-2. Recommendation engine and search as standalone microservices

1 Let's be honest, it's probably tied to some legacy warehouse system you have no power to change.

How do you identify volatile components in your systems? What techniques can you use to identify the best possible microservice candidates?

How Do We Know What Changes Faster Than the Rest?

In this fictitious example, we can simply declare, "These modules change a lot." But that won't fly in the real world. How do you find parts of your application that evolve at different rates? More specifically, how do we find the components that change *far* faster than the rest?

Normally software developers skew logical, but let's use our emotional and rational brains in concert. Odds are, you have an inkling about the part of your app most likely to benefit from faster iteration. Trust your gut instincts!

But you shouldn't rely entirely on feelings. You can use your source code management system to give you a "heat map" of your code. With a Git repository, for instance, you can run `git log` with a few command-line options piped through common Linux tools. You can generate a top ten list of most committed files with a command like this:

```
git log --pretty=format: --name-only | sort | uniq -c
| sort -rg | head -10
```

Don't be surprised when `changelog.txt` shows up at the top of the list! You will have to sanity-check the results, but it gives you a place to start. Now it is time to perform a bit of software archeology. You need to root around your codebase looking for (to paraphrase Isaac Newton) smoother pebbles and prettier shells.[2] This work hearkens back to the concept of churn, first introduced by Michael Feathers. *Churn* is a way of informing decisions on refactoring. When you look at file churn for a given project, you are almost always going to see a long tail distribution. You can visualize what this histogram

2 From *Memoirs of the Life, Writings, and Discoveries of Sir Isaac Newton* by Sir David Brewster (1855): "I do not know what I may appear to the world, but to myself I seem to have been only like a boy playing on the seashore, and diverting myself in now and then finding a smoother pebble or a prettier shell than ordinary, whilst the great ocean of truth lay all undiscovered before me."

looks like—some files change constantly, while others haven't been touched since the initial commit. Based on Feathers's work, Chad Fowler created Turbulence, a visualization into the churn versus complexity of a codebase.

There are also new "code forensics" tools like CodeScene that can yield deeper insights into your projects. CodeScene identifies hot spots in your code, shining a bright light on areas that will be hard to maintain. The results also underscore the parts of your app that could be at risk if a specific developer leaves. You can bring tools like this to bear on your projects, using them to help you identify the prime candidates for a microservice transformation.

But you don't have to add yet another piece of software to your world—just look at the last commit on GitHub. You'll inevitably find that some files were last modified a few moments ago, while others haven't been updated in years. If a file hasn't been touched since the last Super Blue Blood Moon Eclipse, the rate of change factor won't push it into a standalone microservice pattern. But if you see a set of files that appear to always be changing, you should dig deeper in those areas.

Your bug tracker and your project management tools will also help you zero in on likely candidates. Defect density might point you in interesting directions. It also makes sense to review the stories in your backlog. Which modules seem to have a disproportionately high amount of attention? Those are worth exploring.

Using these tools and your instincts, you have an idea what components change more often than others. Now you need to decouple them from the rest of the application. How do you do that?

Fortunately, there's a proven technique for this exact task.

Applying the Strangler Pattern

The *strangler pattern* was introduced by Martin Fowler as a way of handling rewrites of legacy systems.[3] Fowler was inspired by the strangler vines he encountered on a trip to Australia. These vines spread seeds in the branches of fig trees and then gradually work

3 Fowler, "StranglerFigApplication," June 29, 2004.

their way down to the soil, all the while gradually killing their host tree.[4]

Applied to software, the approach suggests that an abrupt "rip and replace" upgrade is fraught with peril. Which it is! Instead, the strangler pattern argues that we should build the new system around the edges of the old, gradually retiring the legacy app over time.

The strangler pattern greatly reduces project risk. Instead of rolling the dice on a big bang cutover, you incrementally improve the application over time. A series of small, easily digestible steps increases your odds of success. Your teams can also deliver business value on a regular cadence (something customers really appreciate), while carefully monitoring progress toward the ultimate goal of monolith retirement.

You can take the strangler pattern a step further with a data-driven approach. That chunk of the app you've identified for refactoring? There's a good chance you don't understand every aspect of what that module does. With years of patches, fixes, and undocumented changes, no one likely knows the entire story of what the system is supposed to do. Rather than risk incomplete understanding—and thereby injecting errors into a critical business system—you can rely on real-world data to guide you.

The data-driven strangler introduces a proxy layer between the client (phone, web browser, another app) and the legacy system. This proxy layer intercepts all requests and responses, logging the results as described in Figure 2-3. The proxy layer pays off for you in two ways. First, it provides vital information about how the current system behaves, giving you request/response data you can use to generate tests. Second, this data allows you to verify the new functionality, ensuring it matches the legacy system.

4 Good metaphor for software projects, no?

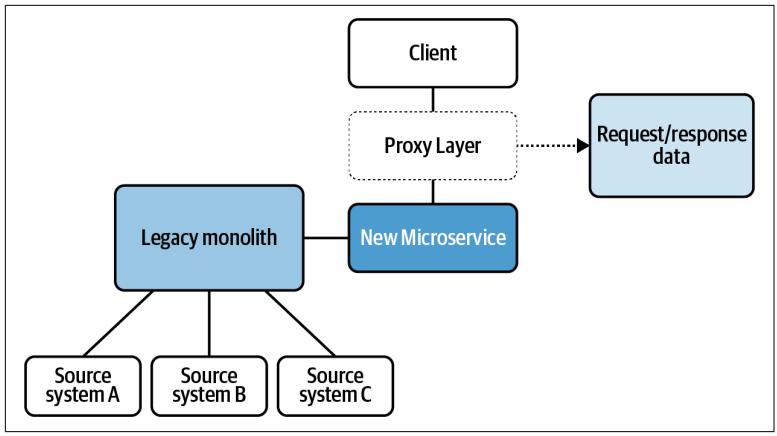

Figure 2-3. Deploy the data-driven strangler to reduce the risk of replacing heritage functionality

In some situations, the new microservice could even be run in parallel with the legacy system, with the proxy layer routing the request to both the heritage monolith and the new microservice comparing the results. If the results match, so much win! But if the results conflict, the proxy can switch over to the legacy system by default, recording the miss for further inspection.[5] Based on the data, you can continue to enhance the new microservice while at the same time adding test cases to your test suite for the new codebase. This approach further improves your confidence in the new code.

To channel Obi-Wan Kenobi, you've taken your first step into a larger world. Your instincts—along with some code archeology—will help you identify multiple rates of change in your systems. Pull out your volatile features as their own microservice. The end result will be simpler code, allowing faster development that's also lower risk. Further, when you apply the strangler pattern, you can confidently maintain existing functionality without introducing new bugs. Not only can these microservices evolve at their own rate, but they can also benefit from independent life cycles.

5 Don't be surprised if the old system is actually wrong!

CHAPTER 3

Independent Life Cycles

For a variety of reasons, you often find parts of your application that need their own commit-to-production flow. Let's consider this dynamic and how microservices can help. *Independent life cycles* are a larger concept that is arguably a superset of *multiple rates of change*, allowing you to be nimbler in today's disrupted marketplace.

Always Be Changing

Monoliths are big ships and they don't turn on a dime. While that might have been workable a long time ago in a galaxy far, far away, today you must be able to respond to an increasingly dynamic environment. Speed matters, and your business partners may not be able to wait for a quarterly release window. Standalone microservices can have their own life cycle with their own repository and a separate deployment pipeline containing the appropriate tests and code quality scans allowing you to capitalize on new opportunities.

The semiannual release cycle doesn't cut it anymore. A monolithic approach usually hinders our ability to deliver quickly, run A/B tests, and learn from users. Chances are, you are being asked to innovate in days or weeks, certainly not months or years. Disruption impacts *every* business today, and your industry is not immune. Returning to the Widget.io example, what happens if your business partners identify a new opportunity? Can you get that functionality into production sooner rather than later? You could tell them to just wait until the next release, but the business opportunity might vanish. Instead, consider the approach described in Figure 3-1. Giving

the new functionality its own life cycle frees it from the schedule tyranny of the slowest-moving part of the application.

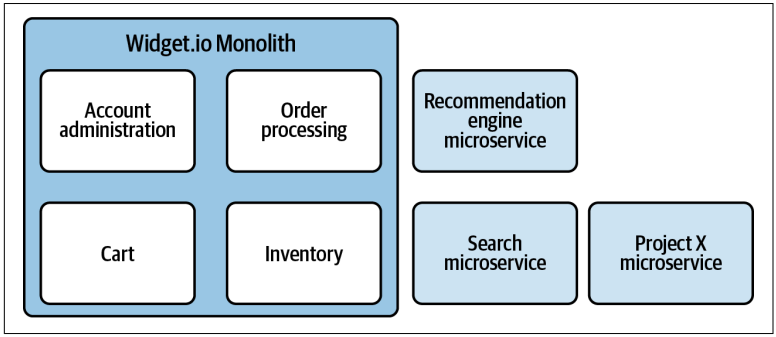

Figure 3-1. Split the Project X functionality into a standalone microservice

In our hypothetical scenario, our business leadership identified a new opportunity that requires speed to market. A typical monolith wouldn't allow us to iterate fast enough, so we made Project X its own microservice. Project X has its own code repository and deployment pipeline, and therefore an independent life cycle. This will help us evolve Project X as we learn more about the business opportunity.

But speed to market isn't the only reason you might want an independent life cycle for a module. It can dramatically increase developer productivity too.

Documenting Services

Documentation, much like the cobbler's children, often gets neglected on software projects. Developers typically prioritize features and functionality.[1] Word processors don't lend themselves to a deployment pipeline, and updating documentation can seem like yet another dollop of busywork. Spring REST Docs takes your handwritten Asciidoctor (or Markdown) text and combines it with autogenerated snippets from Spring MVC Test, WebTestClient, or REST Assured.

1 Besides, the code is self-documenting.

Asciidoctor produces HTML, which you can style and format to your heart's content. Deriving information from tests means your documentation naturally evolves with your codebase. Spring REST Docs lets you focus on describing the requests and responses of your API, allowing you to change implementation details without having to nuke and pave your documentation.

Independent Life Cycles Boost Developer Productivity

Very few developers would say monoliths help them be productive. They slog through dictionary-length developer setup guides. Build times are measured with a sundial. It can take months for a developer to get up to speed on a project. With a smaller scope, a developer can get their head wrapped around a microservice in a day or two. Builds finish in a few minutes (or less). If the build gets broken, engineers know right away and they can take immediate action to fix the issue.

Smaller codebases also mean testing a microservice can be far simpler. Tools like Spring Cloud Contract can help you ensure your services are good citizens and play well with others. You won't be bogged down with the monolith's 80-hour (manual!) regression test suite. Instead, build a set of fine-grained tests against a microservice that can be executed on every commit. Rather than a one-size-fits-none approach to testing, you can bring the right tools and techniques to bear on the individual circumstances of a given microservice. You can subject your microservices to constant scrutiny instead of a one-off performance test. Imagine how this boosts code quality.

From Code to Prod: A Tale of Two Life Cycles

Let's compare monoliths and microservices as they relate to the life cycle; more specifically, how new code goes from a developer's laptop to production.

In the not too distant past, many IT organizations took a singular approach to software development. Projects plodded along in typical waterfall fashion, with quarterly or annual releases. Perhaps a review board (or three) had to sign off before code could go to production.

Seems logical enough, right? But it often led to sleepless nights, long weekends, and windowless war rooms filled with weary people. A shared life cycle meant every module was constrained by whichever one had the longest commit-to-production flow. It also meant every line went through the same process regardless of what stages were most applicable.

Microservices are all about flexibility, including customized deployment pipelines. You are no longer forced to push every line of code through the exact same sieve. In the same way microservices allow you to choose the best technology for the job, you also have the freedom to use the right mix of tests, linters, and code-quality scans for each microservice.

Multiple Owners

While not a technical issue, if your system has multiple independent, autonomous business owners, then it has two distinct sources of change. Unfortunately, this situation often results in conflicts. With microservices, you can achieve independent life cycles and please these different constituencies.

Similarly, if an application is owned by multiple teams, the resulting coordination cost for them working on a single system can be high. Instead, define APIs for them. From there, each team can build an independent microservice using Spring Cloud Contract or Pact for consumer-driven contracts testing.

Fine-grained components—microservices—also make it simpler for you to adhere to your architectural goals. As you refactor your code, it is important that you don't violate a key aspect of your architecture. But how do you ensure that across multiple developers, working in small, independent teams? Fitness functions to the rescue! Arising out of evolutionary computing, fitness functions test the mutation of an algorithm to see if a given change is an improvement or not, ensuring that, over time, the algorithm becomes better and better. As it turns out, this same approach can be used in architecture to allow you to test your architecture as features are added and code is refactored.

For architecture, fitness functions are ultimately about protecting the quality attributes of your system, performance, reliability, simplicity, and so on. Fitness functions are limited only by your imagination, but you might write a test that makes sure service calls respond within a given time frame, that there are no cyclic dependencies in your codebase, or that the number of timeouts doesn't exceed a given threshold. Ideally all of your fitness functions are automated, but don't be surprised if some things have to be manual.

Microservices can make it simpler to find relevant fitness functions. Instead of a one-size-fits-all approach, you can select the proper set of fitness functions for each microservice to ensure that your design evolves in a way that supports key quality attributes.[2]

Hypothesis-Driven Development

Independent life cycles make your life better. They also allow you to make more informed decisions about how your software should evolve. Throughout my career, I have had countless debates with fellow software engineers and customers about possible solutions for various scenarios. And while there were always strong opinions, data was hard to come by. We had to make a decision based on what little we knew and hope for the best.

Of course, even if we were wrong, lengthy deployment cycles meant it would be months before we could alter course. These constraints forced us to be conservative. We couldn't afford to try something unconventional, lest it alienate our users.

> Prediction is very difficult, especially if it's about the future.
>
> —Niels Bohr (attributed)

The scientific method is straightforward. Form a hypothesis based on your observations, then design an experiment to test that theory. What if you could apply a bit of high school science to your software? By using hypothesis-driven development, you can make far better decisions about your software. Independent life cycles make it possible!

2 For more on evolutionary architectures, see *Building Evolutionary Architectures* by Neal Ford, Rebecca Parsons, and Patrick Kua (O'Reilly), and the Building Evolutionary Architectures website.

Taking its cue from a traditional user story, you can formulate something like this:

> We believe <this change>
> Will result in <this outcome>
> We will know we have succeeded when <we see X change in this metric>

For example:

> We believe adding a distributed cache
> Will result in faster startup times
> We will know we have succeeded if startup time is less than 15 seconds

And, you can often turn that structure into a fitness function that you regularly execute against your code.

When a given service has its own commit-to-production flow, you can run multiple experiments reacting to actual results instead of spending countless hours arguing about the future. Today, companies like Google and Amazon run multiple experiments daily (sometimes hourly!). They constantly A/B test. The result: hard data about the impact of a given design on key metrics. What customer doesn't want constantly improving products aligned ever more closely with their needs? More practically, what software organization doesn't want to deliver this kind of service? This is another reason why microservices are so popular.

While it may be easy enough to *want* independent life cycles, it can be daunting to actually achieve it. Regardless of why you are using microservices, you will need a heavy dose of automation in your build process. You need deployment pipelines.

Deployment Pipelines

Modern distributed architectures, along with cloud environments, give you a powerful toolkit to build applications that quickly deliver business value. You can no longer afford a plodding release cycle with nebulous review boards and heavyweight gates slowing development to a crawl. But how can you ensure releases don't bring down production? You need to move fast, but you cannot afford to break things—you need *deployment pipelines*. Leveraging *continuous*

integration (CI) and *continuous delivery* (CD), you can rapidly deliver features and functionality while still getting a good night's sleep. How do you keep your services healthy? How do you know you can trust them? The key is deployment pipelines.

CI and CD to the Rescue

Odds are you've heard of CI and CD, but there is a fair amount of confusion around the topic. What is the difference (other than a single letter)?footnote:[For more on this topic, see the Tanzu developer guide.

In a nutshell, continuous integration could be summarized as "merge early, merge often." But it is more than that. It starts with an automated build separate from the *integrated development environment* (IDE). Code isn't just built, it is also subject to a barrage of automated tests. Ranging from traditional unit tests all the way through various integration or end-to-end tests, automated tests provide a safety net for developers. A robust test suite allows developers to change code at will, while knowing a broken test will alert them to any issues their changes may have caused. Code quality scans can be employed to catch common mistakes and antipatterns, helping to ensure adherence to standards. Again, deviation is detected and fixed early.

Continuous delivery builds upon the foundations of CI, taking it to the next level: releasing code.[3] It takes automation to the release process, allowing you to decide what cadence is most appropriate. The goal? Find issues before your code hits production servers. CD takes the deployable unit coming out of your CI process and moves it from a dev region all the way through to production. This process typically involves a pass through QA and a customer acceptance gate, both of which may involve a manual sign-off.

3 There is some ambiguity about the definition of the *D* in CD. Arguably the most common meaning is as described here. Continuous deployment is one step beyond continuous delivery where changes that successfully pass all the gates of the deployment pipelines are automatically moved to production. Continuous deployment requires a significant investment in testing to ensure changes do not cause havoc in production, and typically involves feature flags and other advanced techniques.

Deployment pipelines give you a well-worn path to production. You can't become an expert at a given task when you only do it once or twice; expertise grows with repetition. Deploy often, and you develop a kind of digital muscle memory. If you only randomly expose your code to unit tests or linting, you can't expect much improvement. But if you subject your code to the same procedures on each and every commit, you develop a process you can trust.

To support these more dynamic environments, companies are increasingly turning to automation to ensure a consistent, repeatable delivery process. While artisanal coffee may make your morning better, an irreplicable build process isn't anyone's definition of a good way to start your day. It turns out that people aren't very good at doing the same things over and over again (see golf). To ensure that the code you deploy to production meets your expectations, it should pass through a rigorous process. Tools like Concourse, Azure DevOps, and Jenkins help you create robust pipelines. You can craft the proper gates, and gain confidence that your code can pass the proverbial gauntlet.

These pipelines were once bespoke one-off endeavors. Today you can leverage projects like Spring Cloud Pipelines or dotnet-pipelines as a starting point. Following an opinionated build/test/stage/prod flow, you can be up and running in your own environment quickly. And you can be sure that your code does what you say it does, thanks to a shortened "idea to production" cycle.

Move Fast and Fix Things

Software development has changed dramatically in recent years, and you cannot afford to say "That's how we've always done it." Applications are evolving rapidly, requiring you to move fast and fix things.[4] Clinging to decades-old processes is a recipe for failure. Thankfully, you have proven patterns and practices to help you accelerate time to market.

Given the need to iterate quickly, independent life cycles may be one of the least-appreciated benefits of a microservice architecture. Looking for parts of your codebase that need their own commit-to-production flow can be an invaluable learning tool.

4 See Vincent Martí's 2015 post, "Move Fast and Fix Things," on the GitHub blog.

Of course, your application's life cycle isn't the only volatile aspect of modern software. Odds are that parts of your system need to scale at different rates, too.

CHAPTER 4

Independent Scalability

Monoliths often force you to make decisions early, when you know the least about the forces acting on your system. Your infrastructure engineers probably asked you how much capacity your application needs, forcing a "Take the worst case and double it" mentality, leading to poor resource utilization. Instead of wild guesses, a microservice approach allows you to more efficiently allocate compute. No longer are you forced into a one-size-fits-none approach; each service can scale independently.

The Monolith Forced Decisions Early—with Incomplete Information

Let's take a quick spin in our hot tub time machine and head back to an era before cloud, microservices, and serverless computing. Servers were homegrown and bespoke, beloved pets if you will. When developers requested a server for a new project, it would take weeks or months for it to become available.

Years ago, one of my peers requested a development database for our architecture team. It was never going to hold a lot of data, it would mostly just act as a repository for some of our architectural artifacts. One would think a small, nonproduction database could be provisioned somewhat quickly. Amazingly, it took an entire year for us to get that simple request fulfilled. Who knows why this was the

case—strange things happen in traditional enterprise IT.[1] But I do know that these kinds of delays incent a raft of undesirable behavior.

Sadly, this experience is familiar to most IT professionals. Extended lead times force you to make capacity decisions far too early. In fact, the very start of a new initiative is when you know the least about what your systems would actually look like under stress. So you'd make an educated guess, based loosely on the worst-case scenario. Then, you'd double it, and add some buffer. (Plus a little more just in case.) In other words, you would just shrug your shoulders as in Figure 4-1.

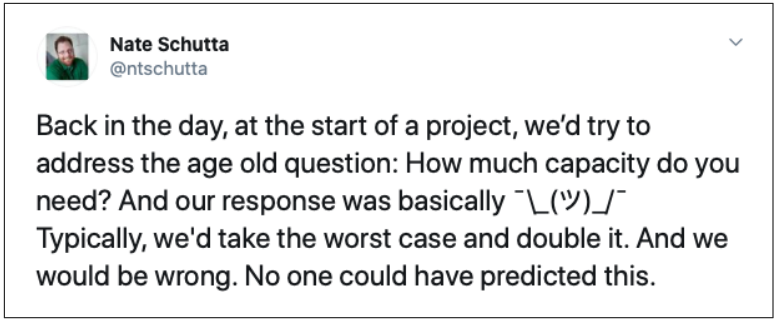

Figure 4-1. How much capacity would you say you need?

And while this meant you often heavily overprovisioned infrastructure (and overspent), that was still a good outcome for organizations. After all, adding capacity after the fact was just as painful.

Any developer familiar with agile development practices understands the disadvantages inherent with big up-front planning and design. Agile ultimately leverages nested feedback loops to give you more information while allowing you to delay decisions to the last responsible moment. Distributed architectures and cloud environments bring many of the same advantages to your infrastructure.

Not All Traffic Is Predictable

It wasn't just your initial server setup that suffered in this era, though. Static infrastructure, combined with your old friend the monolithic application, made it nearly impossible for you to deal

1 I theorize that a team wrote a database as part of the request, but I could be wrong.

with unexpected demand. Even if you had a good understanding of what your systems needed under normal circumstances, you couldn't predict when a new marketing campaign or a shout-out from an influencer would send thousands (or millions) of hungry customers to your (now) overwhelmed servers.

Things were no better for Ops teams running your data centers. It could take months to add additional servers or racks. Traditional budgeting processes made it very difficult to add capacity in any kind of smooth manner. Instead, operators were forced to move in a stepwise fashion, relying on (at best) educated guesses about the shape of future business demand.

While this approach was logical at the time, it meant many organizations had massive amounts of unused compute resources sitting idle nearly all of the time. Single-digit utilization numbers for servers were common. Business units were paying for unused capacity every month just in case a surge or spike in demand occurred. While this may have been an acceptable tradeoff in the past, today it would charitably be considered an antipattern.

Elastic infrastructure combined with more modular architectures mitigate many of these antipatterns. Today, you can add, and just as importantly reduce, capacity on demand. You can start with a reasonable number of application instances, adapting as you learn more about your load characteristics. Working with your business team, you can spin up extra capacity for that big event and then ramp it down after. You can wait until the last responsible moment, freeing you to make better decisions based on real data, not hunches and guesses.

Most recognize this as a benefit of the public cloud, but it's true in the enterprise data center as well. And the savings are just as real. Resources not used by your apps can be leveraged by another group in your organization.

Scale Up Where It Is Needed

The monolith also suffered from a structural constraint: you had to scale the entire thing, not just the bits that actually needed the additional capacity. In reality, the load or throughput characteristics of most systems are not uniform. They have different scaling requirements.

Microservices provide you with a solution: separate these components out into independent microservices. This way, the services can scale at different rates. For example, in Figure 4-2, you are free to spin up additional instances of the Order Processing Microservice to satisfy increased demand, while leaving everything else at a normal run rate.

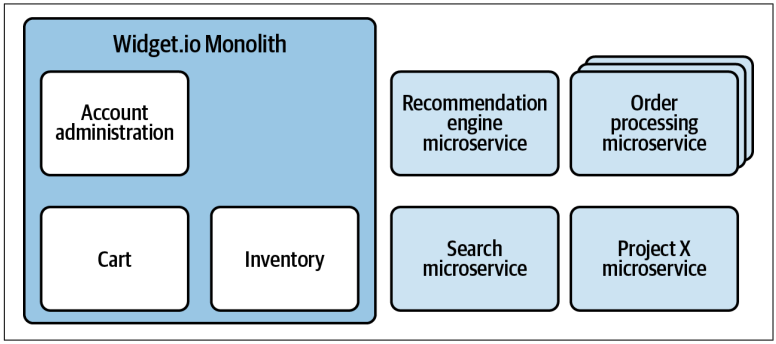

Figure 4-2. Microservices allow you to scale up the parts of your application that need it

In this scenario, it is quite likely that the Order Processing system requires more capacity than the Account Administration functionality. In the past, you had to scale the entire monolith to support your most volatile component. The monolith approach results in higher infrastructure costs because you are forced to overprovision for the worst case scenario of just a portion of your app. If you refactor the Order Processing functionality to a microservice, you can scale up and down as needed.

But how do you know which parts of your application require more capacity? Monitoring to the rescue!

Monitoring for Fun and Profit

Good monitoring is vital for a healthy microservice biome. But knowing what to monitor isn't always obvious. Luckily for us, many organizations are sharing their experiences! You owe it to your services to spend some time perusing *Site Reliability Engineering* by Beyer et al. (O'Reilly), which is filled with wisdom. For example,

Rob Ewaschuk identifies the four golden signals: latency, traffic, errors, and saturation.[2]

The golden signals can indeed provide insights into parts of your system that could benefit from independent scalability. Over time, you will gather priceless intelligence about the actual usage patterns of your application, discovering what is normal (green), what values send a warning about future issues (yellow), and what thresholds are critical, requiring immediate intervention (red).[3] Look for areas with significant traffic or where latency exceeds your requirements. Keeping a close eye on saturation and error rates will also help you find the bottlenecks in your system. Where is your system constrained? Memory, I/O, CPU, or network? Proper monitoring provides these insights.

Spring Cloud Sleuth

With a monolith, tracing a request through the code was relatively straightforward: put in a breakpoint (or 20) and run the application. As soon as you add microservices to the equation, things aren't quite so simple anymore. Now calls bounce between 5 or 10 (or more!) services, making them difficult to trace. Thankfully, Spring Cloud Sleuth can help!

Spring Cloud Sleuth handles all the configuration, including spans (the basic unit of work), sampling, and baggage (key:value pairs stored in the span context). It adds trace and span IDs and instruments the common ingress and egress points (servlet filters, message channels, etc.). It can also generate Zipkin-compatible traces if you'd like. With Spring Cloud Sleuth on the classpath, any Spring Boot app can generate trace data!

Metrics shouldn't be a hand-rolled solution; take advantage of tools like Wavefront, Dynatrace, New Relic, Honeycomb, and others. Spring Boot Actuator supports a number of monitoring systems out of the box. Actuator also includes a number of built-in endpoints

2 For more on this topic, see chapter 6 of *Site Reliability Engineering*, "Monitoring Distributed Systems."

3 If you don't know the scalability characteristics of your app, monitoring is a data-driven way to determine those heuristics.

that can be individually enabled or disabled, and of course you can always add your own.

Don't expect to get your monitoring "right" the first time—you should actively review the metrics you collect to ensure you are getting the best intelligence about your application. Many companies today have teams of site reliability engineers (SREs). SREs help your product teams determine what they should monitor and, as importantly, the sampling frequency. Oversampling doesn't always result in better information. In fact, it may generate too much noise to give you accurate trends. Of course, undersampling is also a possibility. Don't be afraid to tweak your settings until you find the "Goldilocks" level!

Choosing what to monitor can also be tricky. The key service level objectives of your system is a good place to start. Resist the temptation to monitor something simply because it is easy to measure. We are all familiar with, shall we say, less than useful metrics like lines of code. Good metrics give you actionable information about your system.

Retention of Monitoring Data

How long should you keep monitoring data around? The most succinct answer is "It depends," but that isn't very helpful. Don't be too quick to throw away data from "interesting" moments in your application's life cycle. For instance, holiday shopping metrics can be incredibly valuable, providing a starting point for future performance tests. Historical data can be helpful in planning as well.

Last, but not least, monitoring is not just for production. When you monitor staging, as well as lower regions, it validates your monitors. Just as you test your code, you should also ensure your monitors are doing what you expect them to do. Early monitoring not only allows your team to gain much needed familiarity with your toolset, it is also essential for performance and stress testing.[4]

4 Which you are also running early and often, right?

All Services Are Equal (But Some Services Are More Equal than Others)

While you should never overlook the importance of monitoring, don't neglect a less technical part of the equation: a conversation with your business partners. It is vital that you understand the growth rate of your services, and how that's linked to the underlying business. What are the primary business drivers of your services? Do your services need to scale by the number of users or number of orders? What will drive spikes in demand? Take the time to translate this information to the specific services that will be affected.

Don't be afraid to use a humble spreadsheet to help your business partners understand the cost/benefit scenarios inherent with various approaches. For example, scaling order processing might incur additional costs, but that translates to fewer abandoned carts and more happy customers. While you might have to swizzle together some numbers, it can also help you decide how to deploy limited resources.

Don't neglect your data in these conversations. At what rate does your data grow? What type of database solution makes the most sense for your problem domain? Are you read-heavy? Write-heavy? Inquiring minds want to know.

Many businesses plan promotions around holidays or marketing events. Others have predictable spikes in business around certain parts of the year. If you are a retailer, odds are you are familiar with Black Friday and the associated sales, as well as the deluge of customers. Some industries have a year-end or quarterly cadence. Take the time to understand the important dates in your company's calendar and plan accordingly. If you don't know, ask. Again, you will have to determine what services are most affected by these business events. Internalizing the key dates for your organization can mean the difference between delighted customers and former patrons, lost revenue, and poor word of mouth.

It should be obvious that some services are more business critical than others. Using the information you've mined from your business team, you can establish a criticality for a given service. Just as you categorize an outage by severity and scope, you can identify which services are business critical. Returning to our mythical Widget.io Monolith, the ability to process orders is far more

important than the product recommendation service. This fact further reinforces our decision to refactor the Order Processing Service. Apply some due diligence to your services. Which of them could be down for a few hours (or days) with minimal impact? Which ones would the CEO know about instantly in the event of an outage?

Modernize Your Architecture to Use Modern Infrastructure

Independent scalability is one of the most powerful benefits of a microservice approach. The monolith forces you to make decisions at a point in the process when you know the least about what your projects would actually need. This leads to overspending on capacity, and your apps are still vulnerable to spikes in demand. Monolithic architectures force you to scale your applications at a coarse-grained level, further exacerbating the issue. Thankfully, things are much better today.

Modern infrastructure allows you to react in real time—adjusting capacity to match demand at any given moment. Instead of paying every month for the extra capacity you only need once a quarter, you can pay as you go, freeing up valuable resources.

Odds are, your applications would benefit from these capabilities. Proper monitoring along with a solid understanding of your business drivers can help you identify modules you should refactor to microservices. Cloud infrastructure allows you to be proactive while also maximizing the utilization of your hardware resources.

As with the previous principles, you can see once again how microservices give you flexibility. Instead of over-allocating resources for those bits that need it, you can tailor your compute to the needs of *your* application. Doing so can not only save you money, but it also gives you the ability to react to the ebb and flow of demand. From here, it is time to turn our attention to when things don't quite go as planned.

CHAPTER 5

Failure Isolation

In a perfect world, bad things would never happen. Sadly, your applications do not inhabit such a magical realm. To paraphrase a fictional mathematician, failure, uh, finds a way. Should you just surrender to the inevitable chaos and let your customers suffer the ramifications? Of course not! Microservices can be used to isolate a dependency, giving you a natural spot to build in proper failover mechanisms. Performing architectural reviews and leveraging patterns like circuit breaker will help your apps thrive within the mayhem of distributed systems.

No Service Is an Island

Digital products don't live alone. Every piece of tech you use today interacts with something else. The same is true for the custom software you write. Each bit of code works as a part of a larger whole, it's just one piece of a system that executes a business process. The name isn't a coincidence: *microservice* implies interdependence with other services.

Monoliths have plenty of dependencies, too. It's common for monoliths to integrate with an aging third-party application that you don't control. Often you are forced to wire these two systems together with baling twine and duct tape. These engineers weren't masochistic. They did the integration for good business reasons, most likely to provide more complete information or a better user experience.

These integrations are a fact of life in software development. You should expect to have connections between services that were never designed to work together. You should also expect that external services are unlikely to meet your service level objectives.

Fallacies of Distributed Computing

Whether new to microservice architectures or not, software engineers should familiarize themselves with the fallacies of distributed computing. According to L. Peter Deutsch and others at Sun Microsystems, these are assumptions developers repeatedly make:

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

None of these are new—this list dates back many years!—yet many of these mistakes continue to fester today. While you could debate which is the most pernicious, it is clear that not everyone has learned from the past.[1]

When a downstream dependency fails, it's tempting to point the finger at a poorly architected piece of technology. But your customers don't care about excuses. They just want to interact with your software, quickly and easily, then get on with their day.

When you need to protect your systems from failures you can't control, microservices are a great option. Failure domains can reveal fault lines and seams in your applications. Refactoring the functionality in question into microservices allows you to isolate that

1 Speaking of learning from the past, read (or reread) "A Note on Distributed Computing" by Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall at Sun Microsystems Laboratories, Inc.

dependency from the rest of your application. More importantly, you can protect your service level objectives (SLOs) by building in proper failover mechanisms. But how do you know where these failures might happen? By taking the time to perform an architectural review.

Architectural Reviews

Odds are, you have a pretty good idea of what aspects of your system will benefit from failure isolation. But don't assume you know where all the dragons live. Take the time to perform an architectural review. Gather all your subject-matter experts together—developers, architects, and site reliability engineers. You don't need any formal architectural artifacts. Draw up the architecture (a whiteboard works really well). Make sure to ask and answer questions like:

- What systems does the application talk to?
- How do they integrate?
- Is it a direct call or does it go through a proxy layer?
- What availability level can you expect from those systems?

Walk through the architecture. Does everyone have a shared understanding of what the application does? Does everyone understand the requirements? Are you all in agreement as to what talks to what? You'll uncover a lot of details if you ask impertinent questions like:

- What happens when that call fails?
- What is our average response time on that request?
- What would our support team change about the user experience?

You will inevitably find gaps in the broader understanding. That is a feature of this exercise, not a bug! What you thought was a direct call might actually go through a message bus. As you explore the architecture, you will find bottlenecks. It turns out the Wombat service has a lower availability level than you need to provide. Interesting failure cases will result, like when the month's end coincides with a Super Blue Blood Moon, for instance.

The Unbearable Lightness of Dates

Many years ago, I helped build an application for the team that managed all our shared data—office codes, regions, states, that kind of thing. A few months after turning the project over to the data team, my director asked if I could help them with a build break. I was happy to pitch in but pointed out they should review the last change they made, only to discover they hadn't updated the codebase in a few days. Interestingly, that morning a test broke. After investigating, I discovered one of the tests I wrote was bound to fail every seven years. I don't honestly remember if I fixed the test or simply noted it would fail again but it was a reminder of the odd behavior calendars sometimes elicit.

I often reflect on this episode as a reminder of how difficult it can be to imagine all the possible failure points in distributed architectures. It isn't easy. Even trivial mistakes can have far-reaching unintended consequences. We owe it to our customers to explore the "the road less traveled by."[2]

All of this information will give you vital intelligence about where your application might benefit from failure isolation. Refactor away!

Failures Find a Way

You cannot afford to have failures cascading up to your users. Once you've isolated a failure, think about how to react when it occurs. Because it will happen. Do you need to add some redundancy to account for the flakiness of the Wombat system? You might need to have backup services for critical aspects of your system. Should you consider the use of eventual consistency mechanisms, like using Redis to cache data? Data ages at different rates: while a stock price from a few seconds ago isn't useful to a trading desk, it is probably fine for retirement account balances. You need to use proven patterns to ensure your systems are resilient. Odds are, your systems will leverage the circuit breaker pattern.

2 Robert Frost, "The Road Not Taken," 1916.

The Circuit Breaker Pattern

Described by Michael Nygard in his seminal book, *Release It!* (Pragmatic Bookshelf), the circuit breaker pattern is vital for stable systems.[3] Quite simply, a circuit breaker protects a service. It monitors calls to the service. When it sees a certain failure threshold, the breaker is tripped (aka opened), redirecting calls to a configured failover mechanism. That could be an alternative service, a default result, or even an error message. A tripped breaker may also result in an alert to the development team. The circuit breaker can periodically let a call through to see if the service has recovered, resetting if the error threshold is no longer exceeded. It can help to visualize the various states of a circuit breaker, for example Figure 5-1.

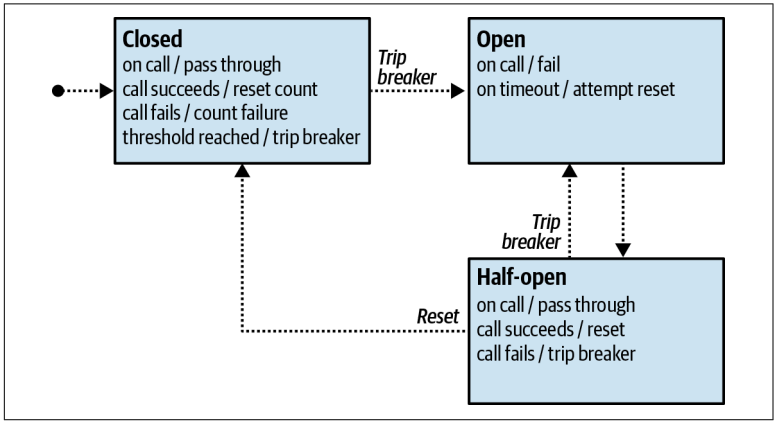

Figure 5-1. An overview of the circuit breaker pattern

Circuit breakers are one of those useful components that simplify building and running microservices. There are multiple options to choose from, like Resilience4j, as well as implementations for virtually any technology stack. Spring Cloud Circuit Breaker provides an abstraction across multiple implementations, giving you a common API and freeing developers to choose the implementation details most suitable to their given application.

3 Don't neglect the other patterns Nygard identifies in his book, such as bulkhead, shed load, and governor.

Spring Cloud Circuit Breaker

Spring Cloud Circuit Breaker provides a consistent API supporting Netflix Hystrix, Resilience4j, Sentinel, and Spring Retry. Simply add the proper starter to your classpath. From there, creating a circuit breaker is just a call to `CircuitBreakerFactory`, which you can then inject into any class as you see fit:

```
@Service
public static class DemoControllerService {
  private RestTemplate rest;
  private CircuitBreakerFactory cbFactory;

  public DemoControllerService(RestTemplate rest,
        CircuitBreakerFactory cbFactory) {
          this.rest = rest;
          this.cbFactory = cbFactory;
  }

  public String slow() {
          return cbFactory.create("slow").run(() ->
        rest.getForObject("/slow", String.class),
        throwable -> "fallback");
  }
}
```

You can customize the behavior of your circuit breaker individually, or create a default configuration for all of your circuit breakers. From here you can configure failure rate thresholds, slow call thresholds, the sliding window size, minimum number of calls, etc. A sample configuration looks something like this:

```
@Bean
public Customizer<Resilience4JCircuitBreakerFactory>
    defaultCustomizer() {

  return factory ->
    factory.configureDefault(id ->
       new Resilience4JConfigBuilder(id)
         .timeLimiterConfig(
           TimeLimiterConfig.custom()
             .timeoutDuration(Duration.ofSeconds(4))
             .build()
         ).circuitBreakerConfig(
           CircuitBreakerConfig.ofDefaults()
         ).build());
}
```

Circuit breakers are vital in a healthy microservice biome. Thankfully, it isn't difficult to add them to your applications. Try it out for yourself, and check out the circuit breaker guide from Spring. Your customers will thank you, and you won't get paged at three in the morning!

As much as you might succeed in predicting where the seams are in your systems, some edge cases are more difficult to divine. To cover your bases, don't be afraid to employ chaos engineering.

Chaos Engineering

Architectural reviews will help you find many soft spots in your system, but they won't identify every instance where you could benefit from failure isolation. Developers tend to be very good at identifying the "happy path" of an application. The "happy path" is the flow a user should experience when everything is working as expected. It's far more challenging to anticipate all the ways your system can go off the rails.

This is exponentially more difficult with distributed applications. A number of services interacting in unpredictable ways leads to unique, often chaotic, environments. How do you ensure a missed failure case doesn't spiral into a major system outage? Chaos engineering to the rescue!

The discipline of chaos engineering attempts to solve the inherent difficulty in producing reliable distributed systems.[4] After defining a steady state (aka normal behavior), chaos engineering injects various issues that real-world systems encounter. Crash an application instance. Simulate a network failure. Drop an availability zone.[5] Failover from one cloud provider to another in production. How does your application handle these situations?

Odds are, at least at first, chaos engineering will highlight some weaknesses with your services. Once again, this result is a feature,

4 For more on this topic, check out Casey Rosenthal and Nora Jones's book, *Chaos Engineering* (O'Reilly), "The Principles of Chaos Engineering," and Adrian Cockcroft's dicussion of chaos architecture. You can also watch presentations from SpringOne Platform 2019 and QCon New York 2017.

5 *Availability zones* protect applications from datacenter failures by providing unique physical locations within a given region. Redundancy is the name of the game, with independent power, networking, and more.

not a bug. Figure out what you need to change in your system to handle the unexpected. Over time, your systems will become more and more reliable. The end result: you and your team will sleep easier at night. Of course, none of this "just happens"; it is the outcome of a significant amount of engineering discipline.

Engineering Discipline

Microservices are a complex architectural option. But they are the right one for certain scenarios, like when you need to isolate failure in certain components. There are ways to reduce the complexity in how you adopt microservices. Architectural reviews, combined with some chaos engineering, will identify vulnerable areas of your current application. Think through how to respond to the inevitable failures. Incorporate circuit breakers where they make sense.

Reliable services aren't a guaranteed outcome of microservices—that requires engineering discipline. Armed with the proper tools and the right approach, your services won't have you questioning your chosen career! In Chapter 6, you will explore a specialization of failure isolation, *indirection layers*.

CHAPTER 6

Indirection Layers

In computer science, there are three answers that work for every question you've ever been asked: "42," "It depends," and "Another layer of indirection." Microservices do not live alone; they work in concert to deliver business value. Your services are constantly working with a veritable cornucopia of other services, all of which are changing and evolving at their own rate (see Chapter 2). How do you protect your services?

This principle is similar to failure isolation (see Chapter 5), but with a twist. Instead of guarding against the inevitable failures of our services, we seek to protect them. "From what?" you ask. From external dependencies that change frequently or are complex to use. This pattern is common in software. Often, this is a vendor dependency, where one service provider is swapped for another. It could be something large (like an ERP system), or something relatively simple (like a mapping service).

Abstract Away External Dependencies

Enterprise systems—monoliths and microservices alike—will inevitably rely on some set of third-party dependencies. Traditionally, your monolith would directly call those other systems or APIs. This was (often) done to ensure acceptable performance. When it came time to change the dependency (for whatever reason), you'd update your system accordingly. Sounds simple, right? It was, but that was a simpler time.

Today, your applications and the systems they often rely on are increasingly complex. When you need to simplify these interactions, microservices can be an appropriate architectural choice.

A microservices architecture offers you a solution. Rather than directly calling these dependencies, you can instead place an abstraction layer (that you control) in between the core app and the dependency. Arguably, this principle is a specialization of failure isolation (Chapter 5), but here we are protecting against a different kind of failure.

Third-party systems are designed to solve a large set of problems, many of which may not be relevant to your application. Instead of forcing your service to understand the nuance of a complex interaction, you can build a microservice that exposes a simplified interface. That's the essence of the *facade pattern*.

This approach is nothing new. In fact, it is one of the 23 original Gang of Four patterns found in the classic book *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma et al. (Addison-Wesley). Facades don't just facilitate evolutionary architecture.[1] They can also be used to simplify a complex service.

That isn't all a microservice facade can do for you, though. Imagine a third-party service that requires a bit of context that doesn't change from invocation to invocation. Perhaps a payment gateway uses your corporate headquarters' location for a calculation, or a vendor requires a token authorizing your use of their API. Rather than have each and every consumer incorporate that bit of business information, your facade microservice can inject the context into the third-party call. Not only does this approach simplify calling the service, but it also makes it far easier to update in the future.

But that's not all! A facade can perform additional functionality before or after making the underlying request. In Chapter 2, you read about the data-driven strangler approach. While obviously useful as part of a larger refactoring, a facade gives you a perfect place to inject needed behavior into your call stack. Want to log the results of the underlying invocation? Or maybe you need to create an audit trail for every call to the third-party system. In either case, your facade microservice makes such interactions trivial.

1 For more on this topic, see *Building Evolutionary Architectures* by Ford et al.

That's the facade pattern in a nutshell! Of course, you might have noticed that your environment is getting a bit more complicated than it was before. How does your application survive in these highly dynamic environments?

Facilitating Legal Compliance

Your applications do not live in a vacuum, and they are increasingly subject to laws and regulations. From General Data Protection Regulation (GDPR) to the Health Insurance Portability and Accountability Act (HIPAA), your applications often deal with sensitive data, which requires compliance with various laws and regulations. Credit cards and personally identifiable information must be safeguarded, and legal departments may need to sign off on your solution. You may be dreading the challenge of refactoring your heritage application, but before you do, consider isolating the sensitive bits in their own microservice. Creating a separate microservice to handle the legally entangled aspects of your application can be simpler and faster than attempting to retrofit the monolith. That isn't to suggest that microservices are a panacea for all legal quagmires, but they can be simpler than the alternatives.

Managing Your Services

As you progress down the path to more distributed applications, you've likely noticed an added complication. Instead of a singular monolith, you now have dozens (perhaps hundreds) of services. Not only are there more services, but they are also changing faster than before. Your services are coming and going as they scale up, scale down, or are replaced. Thankfully, the open source community has a way to help you manage all this thrash: the service registry. Eureka from Netflix is a popular option and as you can probably guess, Spring Cloud allows you to quickly enable and configure service discovery, intelligent routing, client side load balancing, and the aforementioned circuit breaker. Others leverage service discovery within their platform of choice, or take advantage of the capabilities of a service mesh to ease the interaction pain point.

Regardless of which library or approach you use, they are simply implementations of the service discovery pattern. Instead of a brittle, hardcoded configuration, a service registry allows your

application to dynamically find and call services. A typical service registry looks like Figure 6-1.

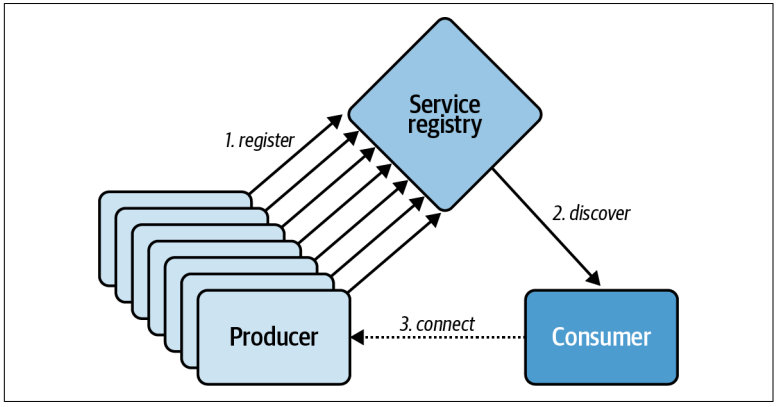

Figure 6-1. Anatomy of a service registry

Clients register themselves providing useful metadata such as host and port. The registry also looks for a regular heartbeat from registered services. If the heartbeat fails, the registry removes that instance to prevent failed calls.

Quite frankly, deploying distributed architectures deserves an in-depth discussion that is beyond the scope of this report. There is no shortage of options at the intersection of development, infrastructure, and application management, along with a significant amount of accidental complexity. There is a reason some refer to these as *death star architectures*. Neglect these issues at your own peril.

At the end of the day, you cannot afford to ignore the architecture of your systems. Whether your organization has an official "architect" role or not, you have people making architectural decisions. I hope for your sake that they are making good ones.

The Importance of Architecture

Architecture is often described as the decisions that are hard to change later, or the choices we wish we'd gotten right early in a project's life cycle. But that definition is often in conflict with the era of disruption. Change is inevitable. And this all-too-common view of architecture has fostered an attitude in many companies that amounts to "we're agile, we don't have architects."

Regardless of titles or roles, you already have people in your organization making architectural decisions.[2] You owe your customers more than just accidental architecture! Microservices can provide additional flexibility, allowing you to refactor easily as new requirements emerge.

You've seen how microservices can help you deal with failures, allow your systems to evolve and scale at their own rates, and how independent life cycles can benefit your applications. Now it is time to turn to the reason many *developers* lobby for microservices: *polyglot technology stacks*.

2 See *Thinking Architecturally* for more on this concept.

CHAPTER 7

Polyglot Technology Stacks

Monoliths forced a standardized, often lowest common denominator, technology stack, regardless of the fit to purpose. Microservices give you the freedom to choose the right language or database for the business requirements rather than force a one-size-fits-all solution.

We're a Java Shop

In the past, companies often tried to standardize on a limited set of technologies. Many development organizations described themselves by their primary technology (i.e., as a Java shop or a .NET team). In the same vein, developers used their favorite language as an adjective, as in "I'm a JavaScript developer."

In the era of the monolith, these colloquialisms made sense. Companies standardized on one tech stack because they could:

- Develop deep expertise with a given language and the associated frameworks
- Shuffle people between teams to balance workloads and cross-pollinate ideas throughout the organization
- Simplify the hiring and training process
- Allow operations teams to specialize in a single environment

All was not puppies and rainbows during this era of software development, though. Things are never that simple—trade-offs are unavoidable. Language standardization often exacerbated currency issues. For example, shared servers lead to months-long slogs to move from version N-2 to version N-1.[1] In the amount of time it took to complete an upgrade, a major new version of the technology was often released. Eighteen months of freezes, testing, change-review boards, and frustration was not the winning recipe for outstanding relationships with your business team.

Bespoke infrastructure often forced the lowest common denominator for library and language versioning as well. Even if a team wanted to move to the latest and greatest version of its preferred tech stack, it may have been limited by the "slowest-moving" heritage application in the house. How often have you heard a variation of, "We can't upgrade to X until the Wombat application is ready for it?" Unless there was a burning platform moment (looking at you, Windows XP), most product owners prioritize shiny new features over paying down technical debt.

All that said, very few organizations were ever truly homogeneous. Mergers and acquisitions happen, inevitably bringing new technologies to the business. Some far-flung requirement would lead to the introduction of a new database or language. But, as a rule, organizations wanted to limit the technology solutions they supported.

Today, you have options; you aren't beholden to a technology choice made by someone five levels above you in the organizational chart.

One Size Fits None

Microservices are a great architectural option when your teams need to choose the right tech for the job. But when does this scenario arise, and why does it matter? It is tempting to let résumé-driven design justify the selection of a new language, but it is a terrible reason to opt for a microservice architecture.

You should consider a new technology stack when there are recognizable benefits to your business. For example:

1 OK, let's be honest: N-7 to N-3.

- New business initiatives are best served with a nonrelational database.
- A different programming language greatly simplifies an algorithm.
- Evolving business requirements suggest an event-driven solution that depends on a distributed streaming platform.

It is tempting to reach for a new toy after reading about a new database or sitting through a webinar on an evolving language. But it is a much more nuanced decision. Can you hire (or train) enough developers with that particular skill set? Before you add complexity to your environment, be sure you have compelling reasons to do so.

Cloud computing opened up a new universe of options and flexibility. Elastic infrastructure combined with microservices architectures enabled you to break free from the tyranny of a singular technology stack. No longer are you required to pound a square peg into a round hole. You can simply pick up a round peg! If a different database simplifies a solution, or if using an alternative language greatly reduces the codebase for a service, you are free to choose the right technology to fit the problem. The promise of polyglot programming is finally realized![2]

However, there is a massive downside to a polyglot approach. To (once again) paraphrase the mathematician in *Jurassic Park*: Just because you can doesn't mean you should. In today's world, your business is evolving at an ever-increasing pace. To keep up, you need to ship high-quality code very quickly. Developers should be free to choose the right tools for the job. So how do you empower your teams responsibly?

Paved Roads

Every developer has their favorite languages, frameworks, and tools. Products will have their own deployment pipelines, monitoring suites, and preferred metrics. Without some guardrails, you will quickly discover there are an awful lot of ways to "do that one thing," depending on who you ask. How do you develop any amount of consistency if nothing is the same?

2 For more on this, see Neal Ford's "Polyglot Programming" post from 2006.

Technical sprawl is only one consequence of a polyglot environment, though. Consider the challenges in building a development team that uses Go, Haskell, Java, .NET, Ruby, Python, and JavaScript. Developers can always learn a new language, but it takes time to start thinking in a given technology. The cognitive overhead of maintaining disparate stacks can be worse than standardizing on a one-size-fits-all model.

And don't underestimate the inherent challenges of maintaining a polyglot environment. For example, think about how much effort it takes to stay current on one stack. Now multiply that toil by four or five, and remember you've signed your development team up for a long-term support agreement on each and every one of those frameworks. Don't forget the ecosystem each stack requires—how much will it cost to support the various logging, monitoring, and deployment approaches as well?

Too much choice can be just as painful.[3] Teams will spend hours (more likely days) "debating" which one to use. Don't be afraid to establish some guardrails or guide posts. For example, you may decide to standardize on the Java virtual machine (JVM), or to provide well-worn paths to production on a limited set of tech stacks. Teams may be allowed to venture off that path, but *they* take on the responsibility (and the support burden) of that decision—you build it, you run it.

> With great power comes great responsibility.
>
> —Uncle Ben, *Spider-Man*

Provide adequate guidance to your teams to help them make the right technology choice. Decision trees are invaluable. Say you have the choice between three different message queues. Create a flowchart asking the relevant questions to help teams narrow down their options. The most agile teams are the most disciplined. This seems counterintuitive at first. But when you think about it, you realize that you gain velocity and responsiveness when you eliminate variation and manual effort. As tempting as it is to let freedom reign, too much choice is as dangerous as too little. Don't be afraid to offer a well-curated menu of options.

3 Side note for all the parents out there: it's much easier to give kids the choice between two or three things rather than a limitless universe of options.

They're Called Microservices

There are as many definitions of microservices as there are companies employing the pattern. But as long as you and your team have a shared understanding of the term within your organization, it doesn't really matter how you define it. However, it is important to stress the *micro* part of microservices.

We can debate what we mean by small, but I've always been partial to the idea that a microservice is a service that can be rewritten in two weeks or less. Emphasizing the small frees us to experiment—and more importantly—to correct course if our hypothesis is proven incorrect.

The more time you've invested in a solution, the less open you are to making changes. How committed are you to code you've spent five minutes on? What about something you spent a few weeks crafting? If you keep your microservices small, you can afford to spend an iteration on an experiment. After all, you've only committed a short amount of time to it. If you are wrong, you can easily adjust.

In the era of the monolith, trying out something new was a recipe for disaster. How could you just mix in a little Clojure/Scala/Groovy/JavaScript into your existing application? With longer development cycles, you might not discover where the "gotchas!" are until it is too late to change course. By focusing on smaller bits of functionality, you have the room to practice hypothesis-driven development on your technology stack as well.

I can't stress this point enough: while polyglot programming might be the most common reason for developers to embrace microservices, you must weigh the pros and cons of a diverse tech stack for your organization. Very few organizations fully adopt the level of developer autonomy that says, "Use whatever you want." But those that do apply the requisite response of, "You build it, you run it." A chef in a professional kitchen brings their own knives to work and is responsible for maintaining the tools of their trade. When a team brings its own technology stack to a project, they, too, are accountable for those decisions. Developers are quick to invent reasons for using shiny new toys. Add new languages and technologies with great care.

Be sure you aren't practicing résumé-driven development. Instead, ask whether this particular language or framework provides demonstrable business advantages, or if you are simply trying to add it to your CV.

CHAPTER 8

The Importance of Culture

What's culture got to do with anything? Well, you ignore the impact of the transition to microservices on your organization's culture at your own peril. While it is tempting to dismiss the "softer" aspects of a new technique, to truly succeed requires more than just writing a strategy statement and adding a three-day class to your educational offerings. You must consider the larger universe that your technology choice lives within.

Culture Impacts Everything

A company's culture is formed *very* early in its existence. People often join, and remain, at an organization because of its culture. How often have you discussed a potential candidate for a position in terms of "culture fit"? While you may not consciously think of it, your corporate culture affects nearly everything, large and small, from what attire is considered appropriate to what kind of snacks are offered in the break room. Because culture is so ingrained, changing it is challenging. People who have risen to power or excelled within the organization have typically learned to "game" the culture and are often resistant to any change that may jeopardize their position.

> It is difficult to get a man to understand something when his salary depends upon his not understanding it.[1]
>
> —Upton Sinclair

In "The Curse of Culture," Ben Thompson writes that "culture is one of a company's most powerful assets right until it isn't," using Microsoft's initial dismissal of the iPhone as a poignant example.[2] He goes on to detail how it took a new CEO to transform the company. While you likely don't have that level of authority, you must be aware of your culture and how it will impact what you are trying to accomplish.

Middle Management Mafia

A good friend of mine spent more than a year helping a retailer modernize their web experience. His work was championed by senior management (praising the project for the millions of dollars worth of sales it drove), and by the developers who were thrilled with the technical elegance his architecture enabled. By any metric, the project was a huge success with easily identified positive effects on the bottom line. However, not everyone was sold on the success. Midlevel managers constantly sniped at the team and often directed their staff to ignore or even subvert the effort. Despite strong support from the top and bottom of the org chart, the project fell victim to what they internally called the "middle management mafia." They failed to see how the project benefited their accountabilities, and actively worked against its success. In many instances, culture is where good ideas go to die.

Changing culture does not happen quickly—it takes patience. It takes years to shift mindsets as well as hire and promote people invested in the new. I spent several years at an organization that made the shift to agile development. Requests for project rooms were met with blank stares and confusion from facilities management. Thankfully, our managers wouldn't take no for an answer, and were continually pressing our case. Eventually, facilities relented,

1 Sinclair, "I, Candidate for Governor, and How I Got Licked," *Oakland Tribune*, December 1934.

2 Thompson, "The Curse of Culture", *Stratechery*, May 24, 2016.

giving us a less than desirable internal conference room lacking natural light or a cell signal. That single room allowed us to show success and build credibility. It also served as a model for future project rooms. Facilities management, freed of the constraints of felt-lined boxes, offered up several innovative suggestions. Eventually the IT floors were all renovated to project rooms of various sizes. It did not happen quickly and took the effort of a great many determined people.

Evolving Your Organization

Sometimes the easiest way to change culture is to start an entirely new one. Some organizations create a special floor (or an entire building) where teams are freed of the normal constraints. Others go further, establishing a group with the express intent to "do things differently," while still others spin out an entirely separate company that is free to build a culture from the ground up. Regardless of the implementation, the goal is the same: give groups of like-minded individuals a fresh start, freeing them from the insidious phrase "That's how we've always done it."

If you've recently started down the microservices path, you may have discovered that your org chart is making things harder than they should be. Conway's Law states:

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.[3]
>
> —Melvin E. Conway

How do you create a series of small, isolated services if your organization isn't made up of small, isolated teams? The Inverse Conway Maneuver tells you to evolve your org chart to match the desired architectural end state. However, you will need to think about how those teams will work together. A microservice is not an island unto itself. What happens when Team A builds something successful and their infrastructure demand (and cost) goes up? Can they "charge back" to the calling applications based on usage? Do they need to rate limit calls? Prepare for conversations with your accountants—

3 Adapted from Conway's paper, "How Do Committees Invent?" published in *Datamation*, April 1968.

you can't afford to ignore your organization's approach to budgeting and capacity planning.

Internal "funny money" is only one piece of the distributed challenge; you must carefully consider how your services change and grow. How do you navigate incompatible change requests from disparate consumers? How does your team prioritize requests? Is it based on traffic? What about business criticality of the system? There are no hard and fast rules; you will need to try out various approaches and adapt to what works best in your world.

By definition, microservices will evolve over time. However, care must be taken to ensure those changes don't break consumers that have come to rely on the service. The answer isn't to pour concrete all over your system preventing change; instead. rely on consumer-driven contracts. In a nutshell, you create a set of tests that "defines" what your service *does* and how it *responds*. Please note, this approach is not a schema, it defines scenarios of usage—given A, return B—and the tests verify the integration point. Provided your changes do not alter the defined contract, you are free to evolve at will.

Spring Cloud Contract

Spring Cloud Contract makes it dead simple to leverage consumer-driven contracts on your projects. Contracts can be written with a Groovy DSL or with YAML. Adding the Spring Cloud Contract Verifier dependency and plug-in to your build file will automatically generate tests when running `./mvnw clean install`. Implement the handling of the requests or messages. Once the tests pass, you can publish the stub artifacts along with your application. These contracts act as living documentation of the expectations of the service! Consumers can also leverage these stubs during integration testing to ensure they are upholding their side of the contract.

Having the proper organizational structure is vital to successfully adopting microservices. Don't try to reinvent the wheel; learn from what has worked (and what hasn't) at other companies. Team Topologies from Matthew Skelton and Manuel Pais is a great place to start.

CHAPTER 9

Migrating to Microservices

Microservices are great! If you need them, that is. The road to microservices is paved with good intentions, but more than a few teams are jumping on the bandwagon without analyzing their needs first. Microservices are powerful, and they should absolutely be in your toolbox. Just make sure you consider the trade-offs. There's no substitute for understanding the business drivers of your applications; this is essential to determining the proper architectural approach. And it turns out, they aren't the only approach you can leverage.

Modular Monoliths, Macro Services, Oh My!

It should be stressed that monoliths span the continuum of modularity and they actually can be structured in such a way that they don't suffer all of the maladies normally associated with the term *monolith*. You can apply microservice design principles to monoliths! When facing all the issues the monolith entails, it is tempting to (attempt) to jump directly to a microservice architecture. However, for many, that path leads to a distributed big ball of mud.

> If you can't build a monolith, what makes you think microservices are the answer?[1]
>
> —Simon Brown

1 "Distributed Big Balls of Mud," *Coding the Architecture* (blog), July 6, 2014.

Obviously, a distributed big ball of mud is the worst of both worlds—all of the accidental complexity, none of the benefits of a truly distributed system. Many have found moving to a modular monolith first facilitates finding bounded contexts within your application. It is possible to break a monolith into modules that give you *some* of the benefits of highly distributed systems. From this new normal, the microservices should be more evident and you can continue your refactoring, stopping when you've reached the proper return on your investment. Your architectural choices span modularity and distributability, as you can see in Figure 9-1.

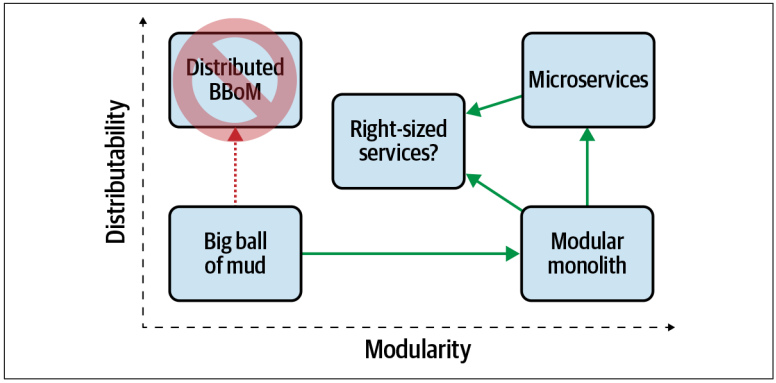

Figure 9-1. Applications exist on a continuum of modularity and distributability

From a modulith to Self-Contained Systems, there are any number of viable architectural approaches. Do what is right for *your* application.

Many adopters of microservices quickly discover they have hundreds, or, more often, thousands of services deployed to production.[2] Microservices are hard! Building, testing, deploying, monitoring, and managing highly distributed systems isn't easy. Some organizations are swinging back to coarser-grained or macroservices. The only right answer is what works best for you.

2 Some organizations' environments are so dynamic, they don't have an exact count of how many services are running at any given moment.

How Big Should a Microservice Be?

As small as possible but no smaller? Small enough to fit inside your head? No more than X lines of code? Many teams want a definitive answer, an algorithm, that will tell you if a service is too large or too small, but, as with much of software, the answer isn't so clear cut. Sam Newman argues that size isn't the right metric as the concept varies by experience and skills.[3] As you build more services, odds are it will be easier for you to, well, build more services. You will have the tools and muscle memory to deploy, manage, and monitor a fleet of services. What you once considered small may now be far too large for comfort. As Sam says, "size" isn't the differentiator here, the real question is how many services can you manage? Do what's right for you and your organization.

Of course, you aren't starting with a blank slate, you have existing applications. How do you decompose the monolith?

Decomposing the Monolith

Software prognostication often ignores the reality facing every organization—the entrenched portfolio of heritage applications that make up the bulk of every company's IT environment. While it is tempting to nuke and pave, you have to face the facts: these applications are powering your business, and you must chart a path forward. If you aren't sure how to start, techniques like event storming can help. Event storming is a collaborative technique designed to help you discover bounded contexts and vertical slices of an application. As a group activity, event storming requires little more than sticky notes, Sharpies, some painter's tape, and a large wall.

As a group, participants "storm the business" process, jotting down domain events on sticky notes. The facilitator will often kick things off by identifying the start and end of the process. Focus on the happy path to begin with and use past tense for events. As your team works through the business domain, you will inevitably find trouble spots, external systems, parallel processing, and time-constrained events like batch processes. Once you've brainstormed the events,

3 Newman, "How Big Should Microservices Be?" (video), uploaded May 3, 2020.

work with domain experts to enforce a timeline, which will often uncover missing elements.

Once you have a timeline, look for domain aggregates, aka bounded contexts. Identify *pivotal* events that transition across subdomains. These clumps of events will often expose candidate services. From here you can also rough out user interfaces, personas, and whatever else is important in the domain.

In addition to event storming, you can apply a set of heuristics to discover domains. Some things to look for:

- The structure of the organization: Where in your organization does the same business concept have different key attributes? For example, an insurance policy means different things to the billing area than it does to the claims department.
- Domain language: Where does a given term mean the same thing and, more importantly, where does it mean something else entirely?
- Where are domain experts positioned in the org chart?
- What is the core domain of the company? Strategic differentiation should inform your breakdown.

Once you have some candidate boundaries, you can test them! Are there any "overloaded contexts"—that is, places where the context does too many things? A multitude of `if` statements indicates you probably have two or more domains. Is your context autonomous? Can it make decisions on its own, or does it need to reach out to a dozen other modules? It may seem a bit fuzzy, but don't forget to do a sanity check. Do these boundaries feel right?

There is a fair amount of art involved when you decompose a monolith—there is no magic formula. Hopefully, these tips will help and give you a place to start your journey. Refactoring takes time, so be patient; your portfolio wasn't built in a day, and you won't move everything to the cloud in a week. Good luck!

Next Steps

I hope this guide has given you some practical advice on *when* to apply microservices.[4] If you want to go deeper, here are some additional resources:

- Neal Ford's "Microservices Essentials" playlist
- *Building Microservices* by Sam Newman (O'Reilly)
- *Production-Ready Microservices* by Susan J. Fowler (O'Reilly)
- *Microservices Patterns Video Edition* by Chris Richardson (Manning Publications)
- *Microservices AntiPatterns and Pitfalls* by Mark Richards (O'Reilly)

There are many good reasons to use a microservices architecture. But there are no free lunches. The positives of microservices come with added complexity. Don't use microservices just to check a box, or because some company you admire leverages them. Do what is right for you and your team.

I hope this report has helped you have more rational and informed discussions about when and how to best use microservices in your enterprise. This topic is one of the most important you'll face in the next five years. Thanks for reading, and please microservice responsibly!

4 And, perhaps more importantly, when *not* to use them.

About the Author

Nathaniel T. Schutta is a software architect focused on cloud computing and building usable applications. A proponent of polyglot programming, Nate has written multiple books and appeared in various videos. Nate is a seasoned speaker, regularly presenting at conferences worldwide, No Fluff Just Stuff symposia, meetups, universities, and user groups. In addition to his day job, Nate is an adjunct professor at the University of Minnesota, where he teaches students to embrace (and evaluate) technical change. Driven to rid the world of bad presentations, Nate coauthored the book *Presentation Patterns* with Neal Ford and Matthew McCullough. Nate recently published *Thinking Architecturally*, available as a free download from VMware.