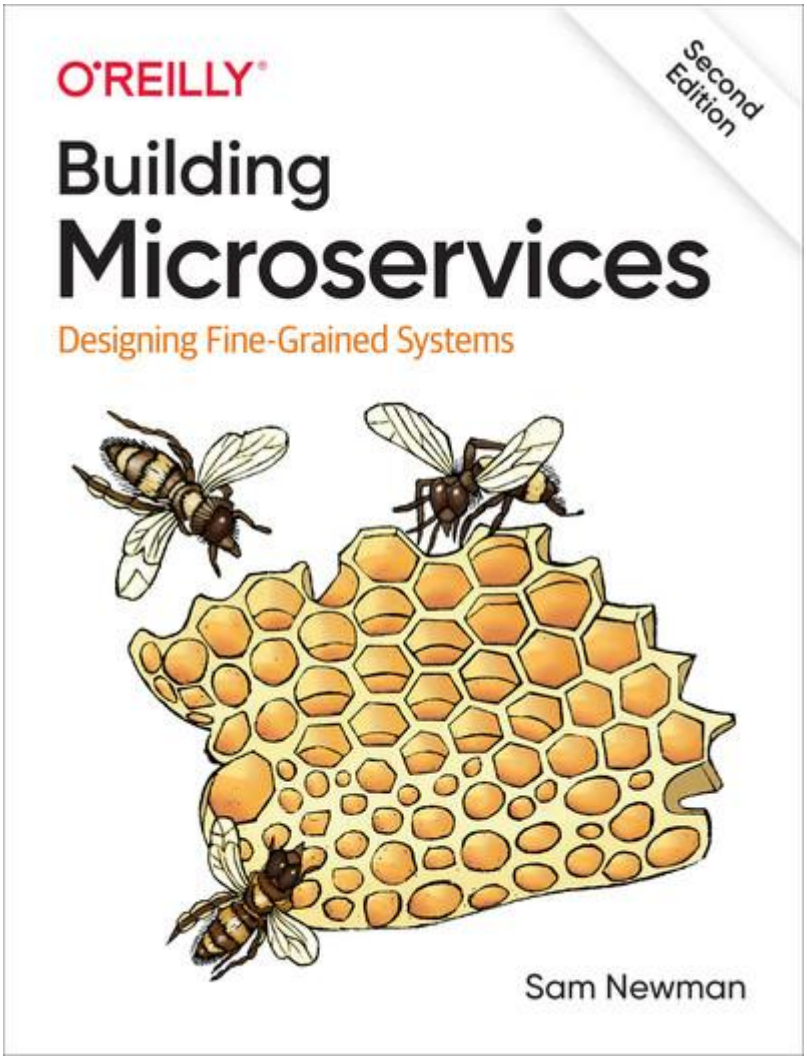

Chapter 1. What Are Microservices?

Microservices have become an increasingly popular architecture choice in the half decade or more since I wrote the first edition of this book. I can't claim credit for the subsequent explosion in popularity, but the rush to make use of microservice architectures means that while many of the ideas I captured previously are now tried and tested, new ideas have come into the mix at the same time that earlier practices have fallen out of favor. So it's once again time to distill the essence of microservice architecture while highlighting the core concepts that make microservices work.

This book as a whole is designed to give a broad overview of the impact that microservices have on various aspects of software delivery. To start us off, this chapter will take a look at the core ideas behind microservices, the prior art that brought us here, and some reasons why these architectures are used so widely.

Microservices at a Glance

Microservices are independently releasable services that are modeled around a business domain. A service encapsulates functionality and makes it accessible to other services via networks—you construct a more complex system from these building blocks. One microservice might represent inventory, another order management, and yet another

shipping, but together they might constitute an entire ecommerce system. Microservices are an architecture choice that is focused on giving you many options for solving the problems you might face.

They are a *type* of service-oriented architecture, albeit one that is opinionated about how service boundaries should be drawn, and one in which independent deployability is key. They are technology agnostic, which is one of the advantages they offer.

From the outside, a single microservice is treated as a black box. It hosts business functionality on one or more network endpoints (for example, a queue or a REST API, as shown in Figure 1-1), over whatever protocols are most appropriate. Consumers, whether they're other microservices or other sorts of programs, access this functionality via these networked endpoints. Internal implementation details (such as the technology the service is written in or the way data is stored) are entirely hidden from the outside world. This means microservice architectures avoid the use of shared databases in most circumstances; instead, each microservice encapsulates its own database where required.

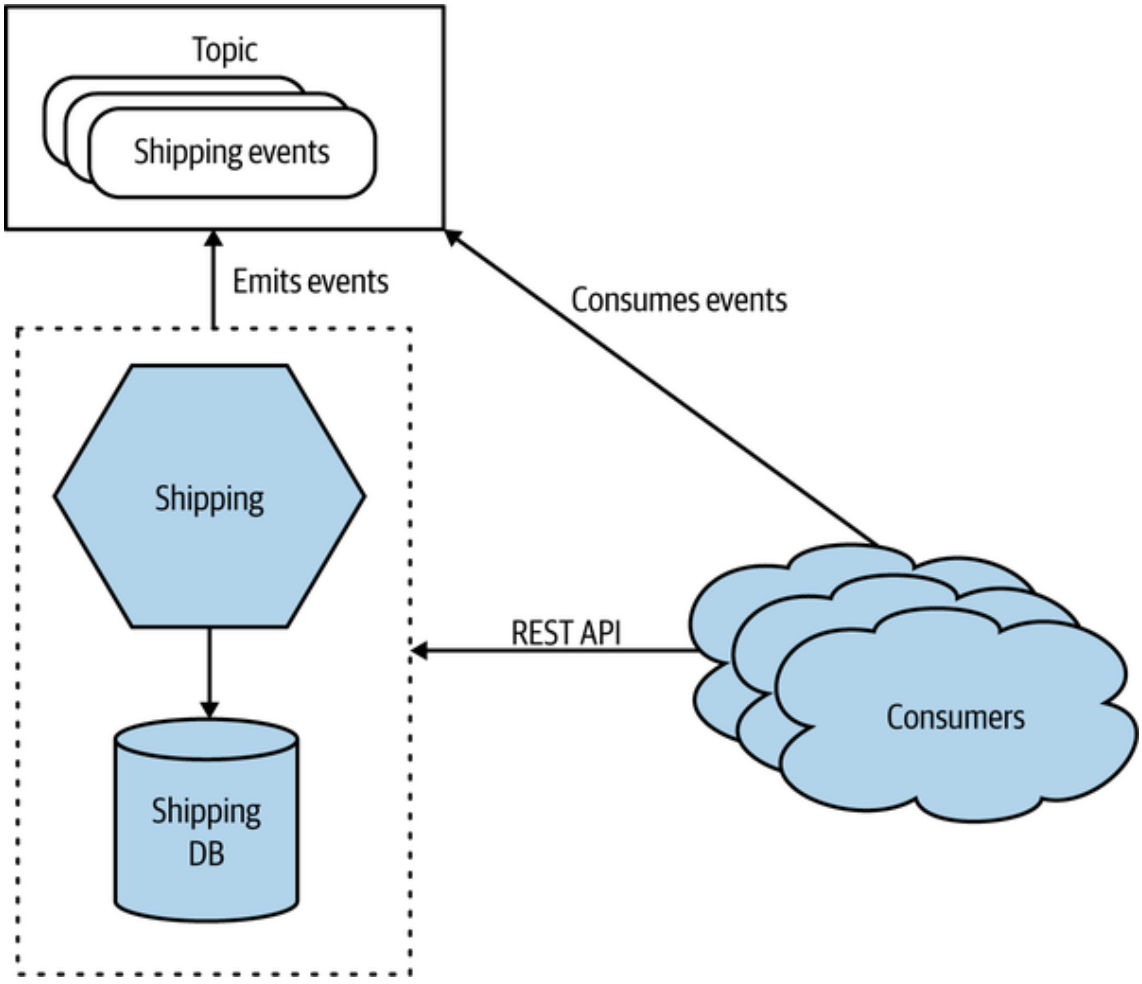

Figure 1-1. A microservice exposing its functionality over a REST API and a topic

Microservices embrace the concept of information hiding.[1] *Information hiding* means hiding as much information as possible inside a component and exposing as little as possible via external interfaces. This allows for clear separation between what can change easily and what is more difficult to change. Implementation that is hidden from external

parties can be changed freely as long as the networked interfaces the microservice exposes don't change in a backward-incompatible fashion. Changes inside a microservice boundary (as shown in Figure 1-1) shouldn't affect an upstream consumer, enabling independent releasability of functionality. This is essential in allowing our microservices to be worked on in isolation and released on demand. Having clear, stable service boundaries that don't change when the internal implementation changes results in systems that have looser coupling and stronger cohesion.

While we're talking about hiding internal implementation detail, it would be remiss of me not to mention the *Hexagonal Architecture* pattern, first detailed by Alistair Cockburn.[2] This pattern describes the importance of keeping the internal implementation separate from its external interfaces, with the idea that you might want to interact with the same functionality over different types of interfaces. I draw my microservices as hexagons partly to differentiate them from "normal" services, but also as an homage to this piece of prior art.

Are Service-Oriented Architecture and Microservices Different Things?

Service-oriented architecture (SOA) is a design approach in which multiple services collaborate to provide a certain end set of capabilities. (A *service* here typically means a completely separate operating system process.) Communication between these services occurs via calls across a network rather than method calls within a process boundary.

SOA emerged as an approach to combat the challenges of large, monolithic applications. This approach aims to promote the reusability of software; two or more end-user applications, for example, could use the same services. SOA aims to make it easier to maintain or rewrite software, as theoretically we can replace one service with another without anyone knowing, as long as the semantics of the service don't change too much.

SOA at its heart is a sensible idea. However, despite many efforts, there is a lack of good consensus on how to do SOA *well*. In my opinion, much of the industry failed to look holistically enough at the problem and present a compelling alternative to the narrative set out by various vendors in this space.

Many of the problems laid at the door of SOA are actually problems with things like communication protocols (e.g., SOAP), vendor middleware, a lack of guidance about service granularity, or the wrong guidance on picking places to split your system. A cynic might suggest that vendors co-opted (and in some cases drove) the SOA movement as a way to sell more products, and those selfsame products in the end undermined the goal of SOA.

I've seen plenty of examples of SOA in which teams were striving to make the services smaller, but they still had everything coupled to a database and had to deploy everything together. Service oriented? Yes. But it's not microservices.

The microservice approach has emerged from real-world use, taking our better understanding of systems and architecture to do SOA well. You should think of microservices as being a specific approach for SOA in the same way that Extreme Programming (XP) or Scrum is a specific approach for Agile software development.

Key Concepts of Microservices

A few core ideas must be understood when you are exploring microservices. Given that some aspects are often overlooked, it's vital to explore these concepts further to help ensure that you understand just what it is that makes microservices work.

Independent Deployability

Independent deployability is the idea that we can make a change to a microservice, deploy it, and release that change to our users, without having to deploy any other microservices. More important, it's not just the fact that we can do this; it's that this is *actually* how you manage deployments in your system. It's a discipline you adopt as your default release approach. This is a simple idea that is nonetheless complex in execution.

Tip

If you take only one thing from this book and from the concept of microservices in general, it should be this: ensure that you embrace the concept of independent deployability of your microservices. Get into the habit of deploying and releasing changes to a single microservice into production without having to deploy anything else. From this, many good things will follow.

To ensure independent deployability, we need to make sure our microservices are *loosely coupled*: we must be able to change one service without having to change anything else. This means we need explicit, well-defined, and stable contracts between services. Some implementation choices make this difficult—the sharing of databases, for example, is especially problematic.

Independent deployability in and of itself is clearly incredibly valuable. But to achieve independent deployability, there are so many other things you have to get right that in turn have their own benefits. So you can also see the focus on independent deployability as a forcing function—by focusing on this as an outcome, you'll achieve a number of ancillary benefits.

The desire for loosely coupled services with stable interfaces guides our thinking about how we find our microservice boundaries in the first place.

Modeled Around a Business Domain

Techniques like domain-driven design can allow you to structure your code to better represent the real-world domain that the software operates in.[3] With microservice architectures, we use this same idea to define our service boundaries. By modeling services around business domains, we can make it easier to roll out new functionality and to recombine microservices in different ways to deliver new functionality to our users.

Rolling out a feature that requires changes to more than one microservice is expensive. You need to coordinate the work across each service (and potentially across separate teams) and carefully manage the order in which the new versions of these services are deployed. That takes a lot more work than making the same change inside a single service (or inside a monolith, for that matter). It therefore follows that we want to find ways to make cross-service changes as infrequent as possible.

I often see layered architectures, as typified by the three-tiered architecture in Figure 1-2. Here, each layer in the architecture represents a different service boundary, with each service boundary based on related technical functionality. If I need to make a change to

just the presentation layer in this example, that would be fairly efficient. However, experience has shown that changes in functionality typically span multiple layers in these types of architectures—requiring changes in presentation, application, and data tiers. This problem is exacerbated if the architecture is even more layered than the simple example in Figure 1-2; often each tier is split into further layers.

By making our services end-to-end slices of business functionality, we ensure that our architecture is arranged to make changes to business functionality as efficient as possible. Arguably, with microservices we have made a decision to prioritize high cohesion of business functionality over high cohesion of technical functionality.

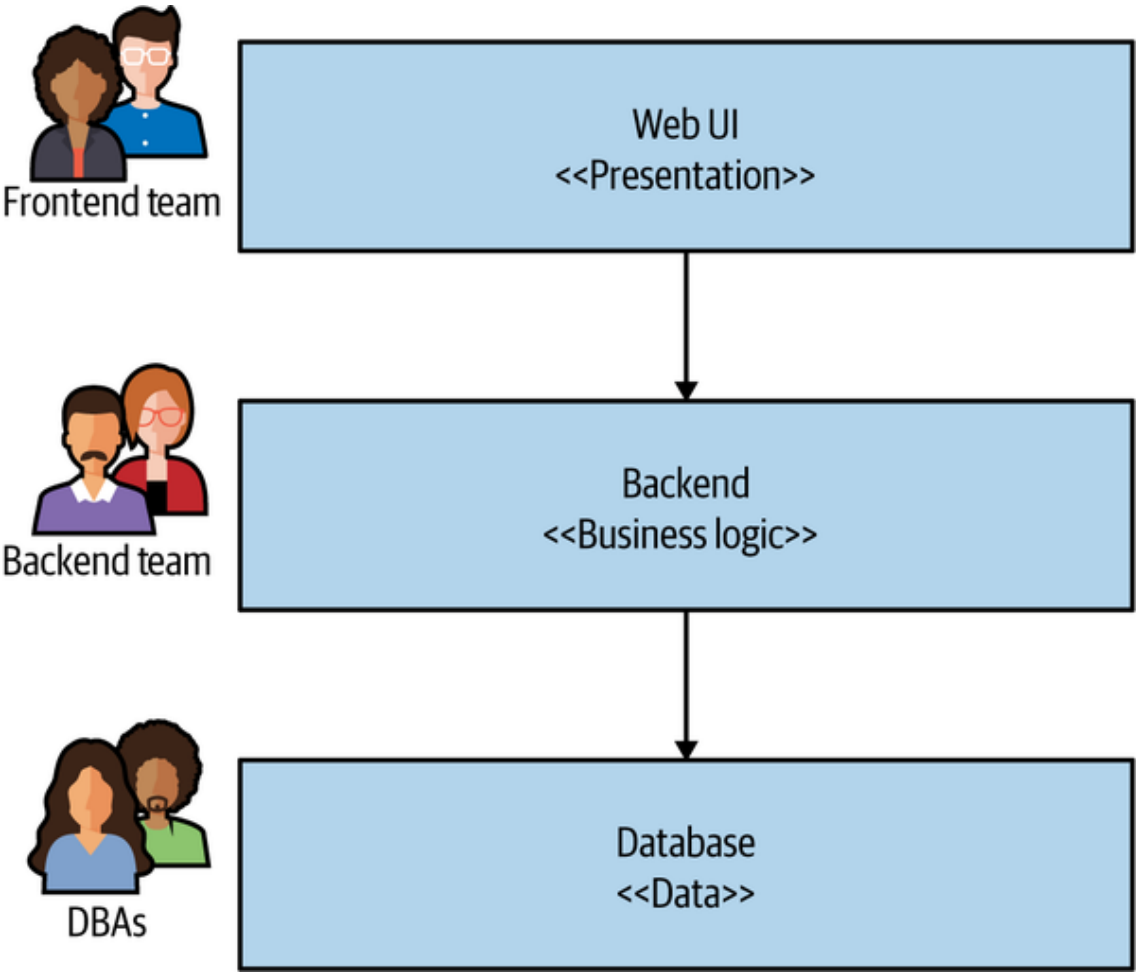

Figure 1-2. A traditional three-tiered architecture

We will come back to the interplay of domain-driven design and how it interacts with organizational design later in this chapter.

Owning Their Own State

One of the things I see people having the hardest time with is the idea that microservices should avoid the use of shared databases. If a microservice wants to access data held by another microservice, it should go and ask that second microservice for the data. This gives the microservices the ability to decide what is shared and what is hidden, which allows us to clearly separate functionality that can change freely (our internal

implementation) from the functionality that we want to change infrequently (the external contract that the consumers use).

If we want to make independent deployability a reality, we need to ensure that we limit backward-incompatible changes to our microservices. If we break compatibility with upstream consumers, we will force them to change as well. Having a clean delineation between internal implementation detail and an external contract for a microservice can help reduce the need for backward-incompatible changes.

Hiding internal state in a microservice is analogous to the practice of encapsulation in object-oriented (OO) programming. Encapsulation of data in OO systems is an example of information hiding in action.

Tip

Don't share databases unless you really need to. And even then do everything you can to avoid it. In my opinion, sharing databases is one of the worst things you can do if you're trying to achieve independent deployability.

As discussed in the previous section, we want to think of our services as end-to-end slices of business functionality that, where appropriate, encapsulate user interface (UI), business logic, and data. This is because we want to reduce the effort needed to change business-related functionality. The encapsulation of data and behavior in this way gives us high cohesion of business functionality. By hiding the database that backs our service, we also ensure that we reduce coupling. We come back to coupling and cohesion in Chapter 2.

Size

"How big should a microservice be?" is one of the most common questions I hear. Considering the word "micro" is right there in the name, this comes as no surprise. However, when you get into what makes microservices work as a type of architecture, the concept of size is actually one of the least interesting aspects.

How do you measure size? By counting lines of code? That doesn't make much sense to me. Something that might require 25 lines of code in Java could be written in 10 lines of Clojure. That's not to say Clojure is better or worse than Java; some languages are simply more expressive than others.

James Lewis, technical director at Thoughtworks, has been known to say that "a microservice should be as big as my head." At first glance, this doesn't seem terribly helpful. After all, how big is James's head exactly? The rationale behind this statement is that a microservice should be kept to the size at which it can be easily understood. The challenge, of course, is that different people's ability to understand something isn't always the same, and as such you'll need to make your own judgment regarding what size works for you. An experienced team may be able to better manage a larger codebase than another team could. So perhaps it would be better to read James's quote here as "a microservice should be as big as *your* head."

I think the closest I get to "size" having any meaning in terms of microservices is something that Chris Richardson, the author of *Microservice Patterns* (Manning Publications), once said—the goal of microservices is to have "as small an interface as

possible." That aligns with the concept of information hiding again, but it does represent an attempt to find meaning in the term "microservices" that wasn't there initially. When the term was first used to define these architectures, the focus, at least initially, was not specifically on the size of the interfaces.

Ultimately, the concept of size is highly contextual. Speak to a person who has worked on a system for 15 years, and they'll feel that their system with 100,000 lines of code is really easy to understand. Ask the opinion of someone brand-new to the project, and they'll feel it's much too big. Likewise, ask a company that has just embarked on its microservice transition and has perhaps 10 or fewer microservices, and you'll get a different answer than you would from a similar-sized company for which microservices have been the norm for many years and that now has hundreds.

I urge people not to worry about size. When you are first starting out, it's much more important that you focus on two key things. First, how many microservices can you handle? As you have more services, the complexity of your system will increase, and you'll need to learn new skills (and perhaps adopt new technology) to cope with this. A move to microservices will introduce new sources of complexity, with all the challenges this can bring. It's for this reason that I am a strong advocate for incremental migration to a microservice architecture. Second, how do you define microservice boundaries to get the most out of them, without everything becoming a horribly coupled mess? These topics are much more important to focus on when you start your journey.

Flexibility

Another quote from James Lewis is that "microservices buy you options." Lewis was being deliberate with his words—they *buy* you *options*. They have a cost, and you must decide whether the cost is worth the options you want to take up. The resulting flexibility on a number of axes—organizational, technical, scale, robustness—can be incredibly appealing.

We don't know what the future holds, so we'd like an architecture that can theoretically help us solve whatever problems we might face down the road. Finding a balance between keeping your options open and bearing the cost of architectures like this can be a real art.

Think of adopting microservices as less like flipping a switch, and more like turning a dial. As you turn up the dial, and you have more microservices, you have increased flexibility. But you likely ramp up the pain points too. This is yet another reason I strongly advocate incremental adoption of microservices. By turning up the dial gradually, you are better able to assess the impact as you go, and to stop if required.

Alignment of Architecture and Organization

MusicCorp, an ecommerce company that sells CDs online, uses the simple three-tiered architecture shown earlier in Figure 1-2. We've decided to move MusicCorp kicking and screaming into the 21st century, and as part of that, we're assessing the existing system architecture. We have a web-based UI, a business logic layer in the form of a monolithic backend, and data storage in a traditional database. These layers, as is common, are owned by different teams. We'll be coming back to the trials and tribulations of MusicCorp throughout the book.

We want to make a simple update to our functionality: we want to allow our customers to specify their favorite genre of music. This update requires us to change the UI to show the genre choice UI, the backend service to allow for the genre to be surfaced to the UI and for the value to be changed, and the database to accept this change. These changes will need to be managed by each team and deployed in the correct order, as outlined in Figure 1-3.

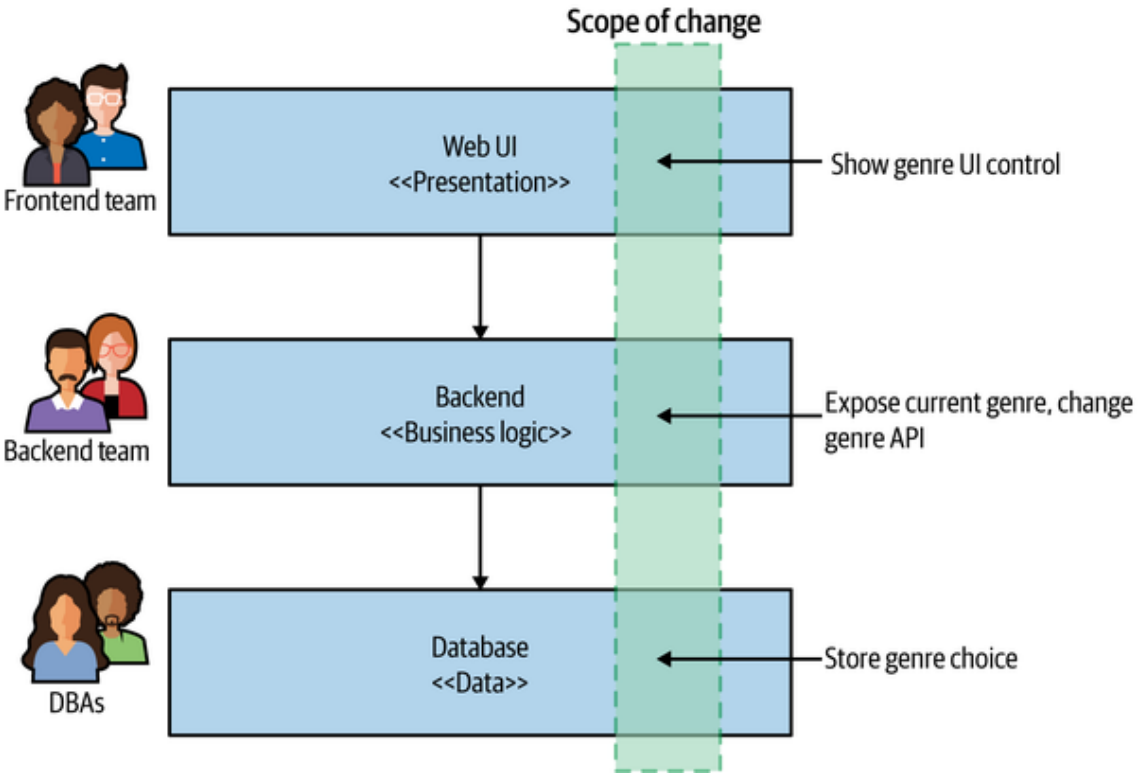

Figure 1-3. Making a change across all three tiers is more involved

Now this architecture isn't bad. All architecture ends up getting optimized around a set of goals. Three-tiered architecture is so common partly because it is universal—everyone has heard about it. So the tendency to pick a common architecture that you might have seen elsewhere is often one reason we keep seeing this pattern. But I think the biggest reason we see this architecture again and again is because it is based on how we organize our teams.

The now famous Conway's law states the following:

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

Melvin Conway, "How Do Committees Invent?"

The three-tiered architecture is a good example of this law in action. In the past, the primary way IT organizations grouped people was in terms of their core competency: database admins were in a team with other database admins; Java developers were in a team with other Java developers; and frontend developers (who nowadays know exotic things like JavaScript and native mobile application development) were in yet another team. We group people based on their core competency, so we create IT assets that can be aligned to those teams.

That explains why this architecture is so common. It's not bad; it's just optimized around one set of forces—how we traditionally grouped people, around familiarity. But the forces have changed. Our aspirations around our software have changed. We now group people in poly-skilled teams to reduce handoffs and silos. We want to ship software much more quickly than ever before. That is driving us to make different choices about the way we organize our teams, so that we organize them in terms of the way we break our systems apart.

Most changes that we are asked to make to our system relate to changes in business functionality. But in Figure 1-3, our business functionality is in effect spread across all three tiers, increasing the chance that a change in functionality will cross layers. This is an architecture that has high cohesion of related technology but low cohesion of business functionality. If we want to make it easier to make changes, instead we need to change how we group code, choosing cohesion of business functionality rather than technology. Each service may or may not end up containing a mix of these three layers, but that is a local service implementation concern.

Let's compare this with a potential alternative architecture, illustrated in Figure 1-4. Rather than a horizontally layered architecture and organization, we instead break down our organization and architecture along vertical business lines. Here we see a dedicated team that has full-end-to-end responsibility for making changes to aspects of the customer profile, which ensures that the scope of change in this example is limited to one team.

As an implementation, this could be achieved through a single microservice owned by the profile team that exposes a UI to allow customers to update their information, with state of the customer also stored within this microservice. The choice of a favorite genre is associated with a given customer, so this change is much more localized. In Figure 1-5, we also show the list of available genres being fetched from a Catalog microservice, something that would likely already be in place. We also see a new Recommendation microservice accessing our favorite genre information, something that could easily follow in a subsequent release.

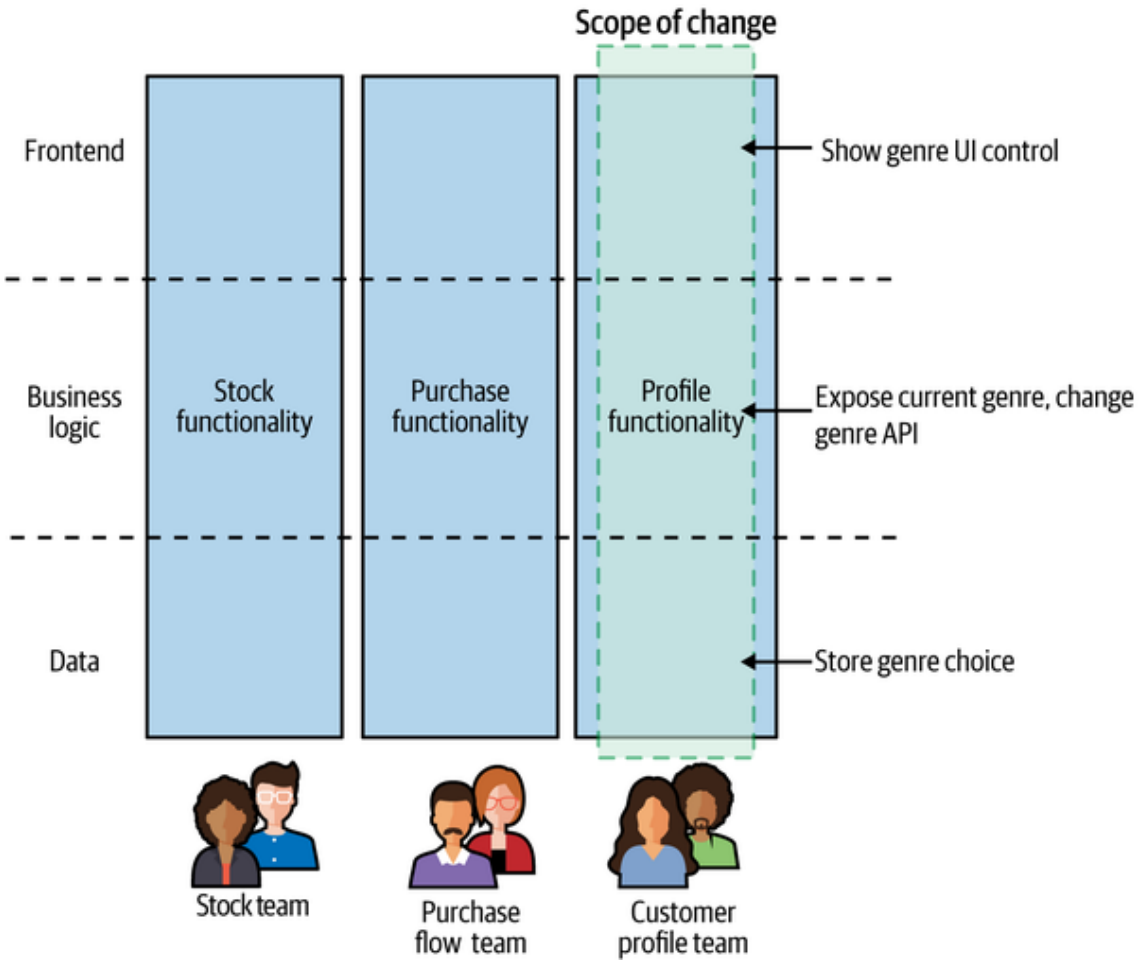

Figure 1-4. The UI is broken apart and is owned by a team that also manages the serverside functionality that supports the UI

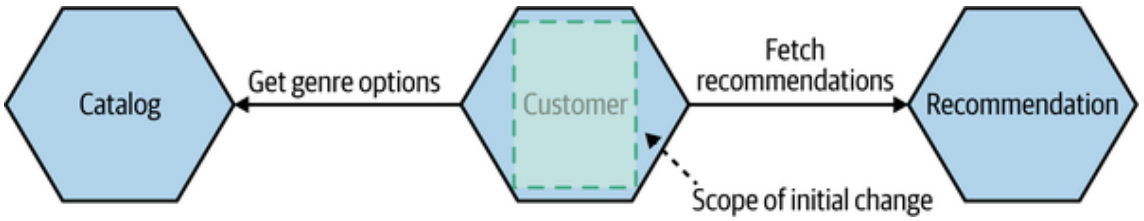

Figure 1-5. A dedicated Customer microservice can make it much easier to record the favorite musical genre for a customer

In such a situation, our Customer microservice encapsulates a thin slice of each of the three tiers—it has a bit of UI, a bit of application logic, and a bit of data storage. Our business domain becomes the primary force driving our system architecture, hopefully making it easier to make changes, as well as making it easier for us to align our teams to lines of business within the organization.

Often, the UI is not provided directly by the microservice, but even if this is the case, we would expect the portion of the UI related to this functionality to still be owned by the Customer Profile Team, as Figure 1-4 indicates. This concept of a team owning an end-to-end slice of user-facing functionality is gaining traction. The book *Team Topologies*[4] introduces the idea of a stream-aligned team, which embodies this concept:

A stream-aligned team is a team aligned to a single, valuable stream of work…[T]he team is empowered to build and deliver customer or user value as quickly, safely, and independently as possible, without requiring hand-offs to other teams to perform parts of the work.

The teams shown in Figure 1-4 would be stream-aligned teams, a concept we'll explore in more depth in Chapters 14 and 15, including how these types of organizational structures work in practice, and how they align with microservices.

A Note on "Fake" Companies

Throughout the book, at different stages, we will meet MusicCorp, FinanceCo, FoodCo, AdvertCo, and PaymentCo.

FoodCo, AdvertCo, and PaymentCo are real companies whose names I have changed for confidentiality reasons. In addition, when sharing information about these companies, I have often omitted certain details to provide more clarity. The real world is often messy. I've always strived, though, to remove only extraneous detail that wouldn't be helpful, while still ensuring that the underlying reality of the situation remains.

MusicCorp, on the other hand, is a fake company that is a composite of many organizations I have worked with. The stories I share around MusicCorp are reflections of real things I have seen, but they haven't all happened to the same company!

The Monolith

We've spoken about microservices, but microservices are most often discussed as an architectural approach that is an alternative to monolithic architecture. To more clearly distinguish the microservice architecture, and to help you better understand whether microservices are worth considering, I should also discuss what exactly I mean by *monoliths*.

When I talk about monoliths throughout this book, I am primarily referring to a unit of deployment. When all functionality in a system must be deployed together, I consider it a monolith. Arguably, multiple architectures fit this definition, but I'm going to discuss those I see most often: the single-process monolith, the modular monolith, and the distributed monolith.

The Single-Process Monolith

The most common example that comes to mind when discussing monoliths is a system in which all of the code is deployed as a *single process*, as in Figure 1-6. You may have multiple instances of this process for robustness or scaling reasons, but fundamentally all the code is packed into a single process. In reality, these single-process systems can be simple distributed systems in their own right because they nearly always end up reading data from or storing data into a database, or presenting information to web or mobile applications.

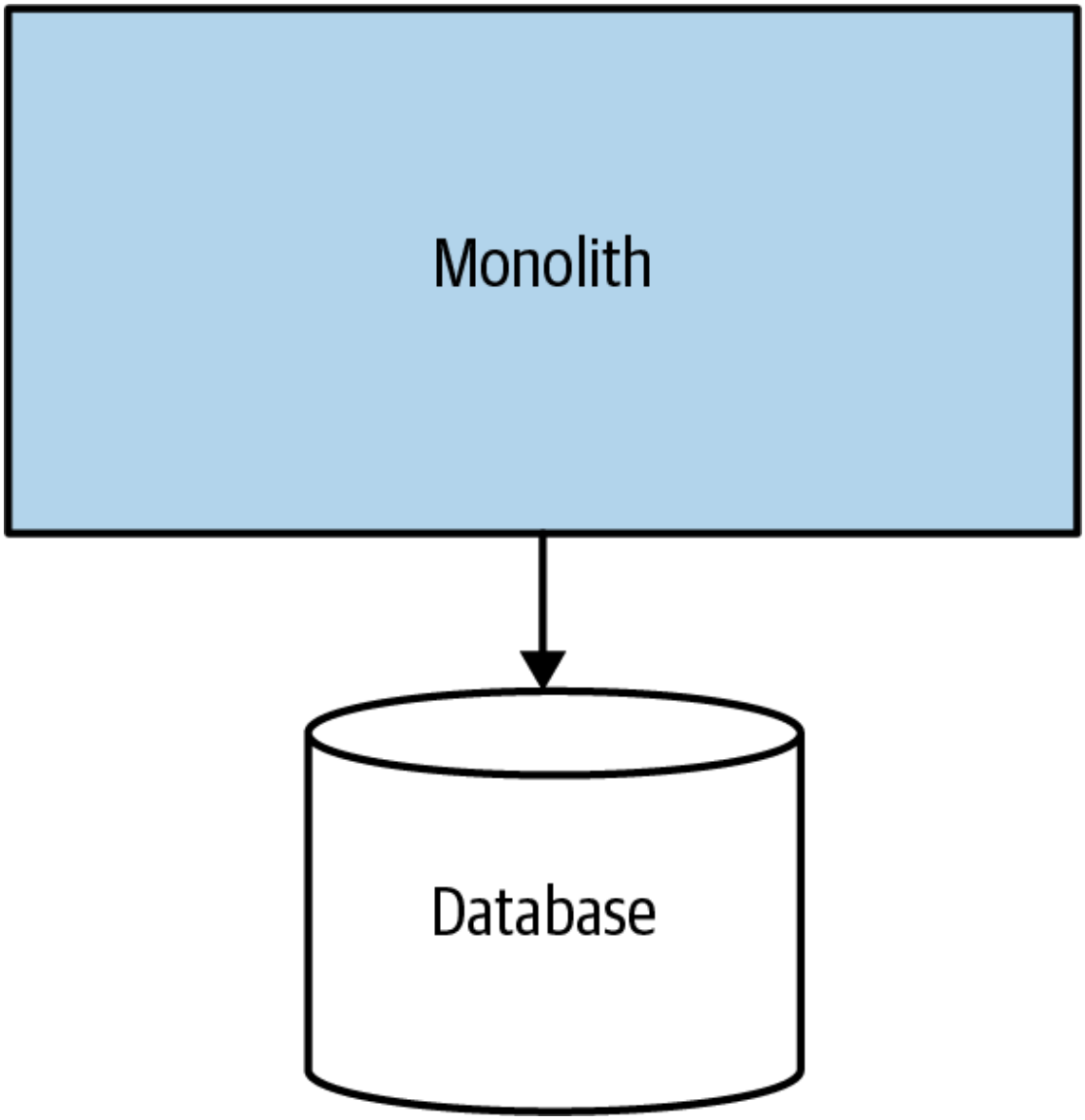

Figure 1-6. In a single-process monolith, all code is packaged into a single process

Although this fits most people's understanding of a classic monolith, most systems I encounter are somewhat more complex than this. You may have two or more monoliths that are tightly coupled to one another, potentially with some vendor software in the mix.

A classic single-process monolithic deployment can make sense for many organizations. David Heinemeier Hansson, the creator of Ruby on Rails, has made the case effectively that such an architecture makes sense for smaller organizations.[5] Even as the organization grows, however, the monolith can potentially grow with it, which brings us to the modular monolith.

The Modular Monolith

As a subset of the single-process monolith, the *modular monolith* is a variation in which the single process consists of separate modules. Each module can be worked on independently, but all still need to be combined together for deployment, as shown in Figure 1-7. The concept of breaking software into modules is nothing new; modular software has its roots in work done around structured programming in the 1970s, and even

further back than that. Nonetheless, this is an approach that I still don't see enough organizations properly engage with.

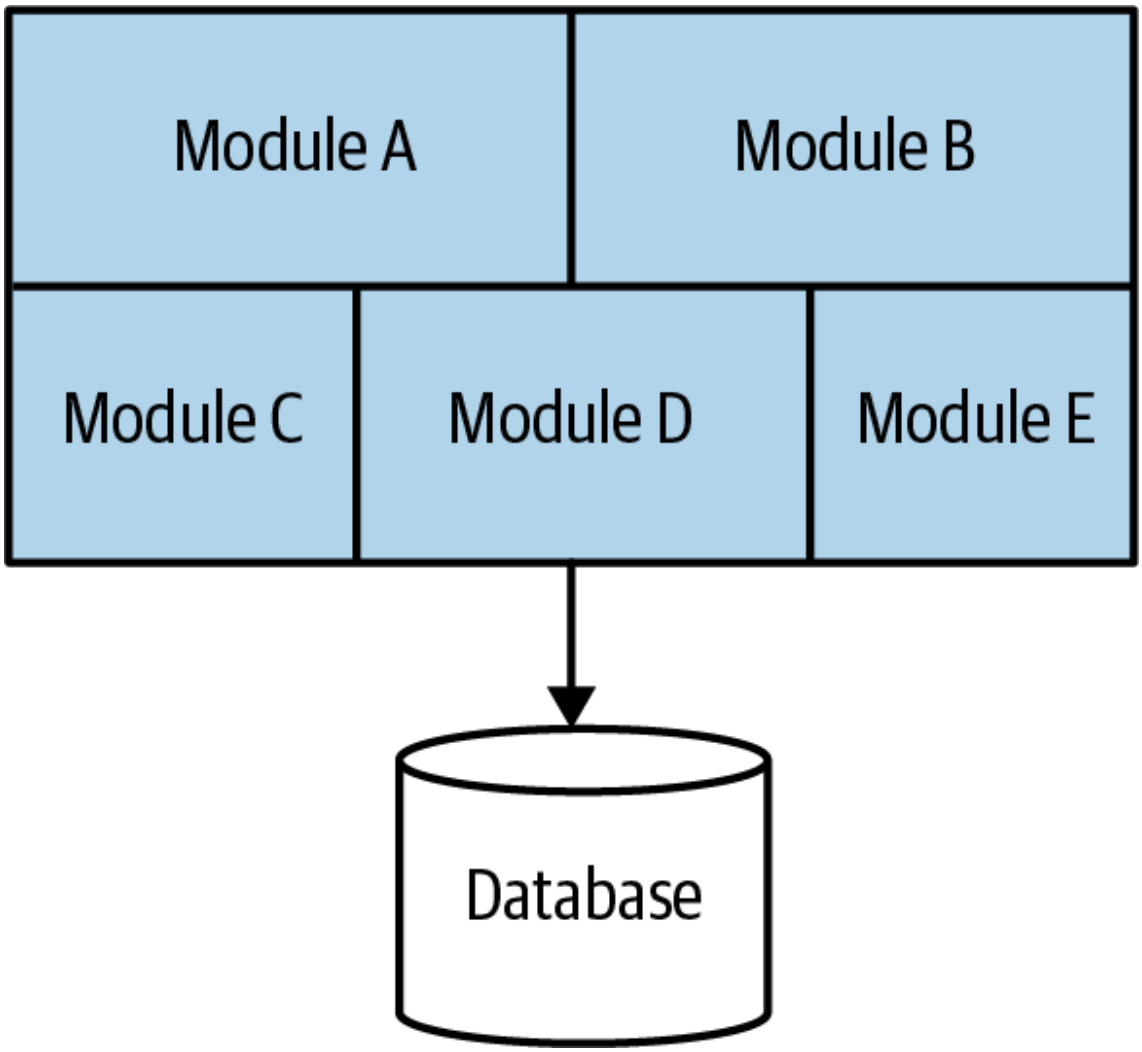

Figure 1-7. In a modular monolith, the code inside the process is divided into modules

For many organizations, the modular monolith can be an excellent choice. If the module boundaries are well defined, it can allow for a high degree of parallel work, while avoiding the challenges of the more distributed microservice architecture by having a much simpler deployment topology. Shopify is a great example of an organization that has used this technique as an alternative to microservice decomposition, and it seems to work really well for that company.[6]

One of the challenges of a modular monolith is that the database tends to lack the decomposition we find in the code level, leading to significant challenges if you want to pull apart the monolith in the future. I have seen some teams attempt to push the idea of the modular monolith further by having the database decomposed along the same lines as the modules, as shown in Figure 1-8.

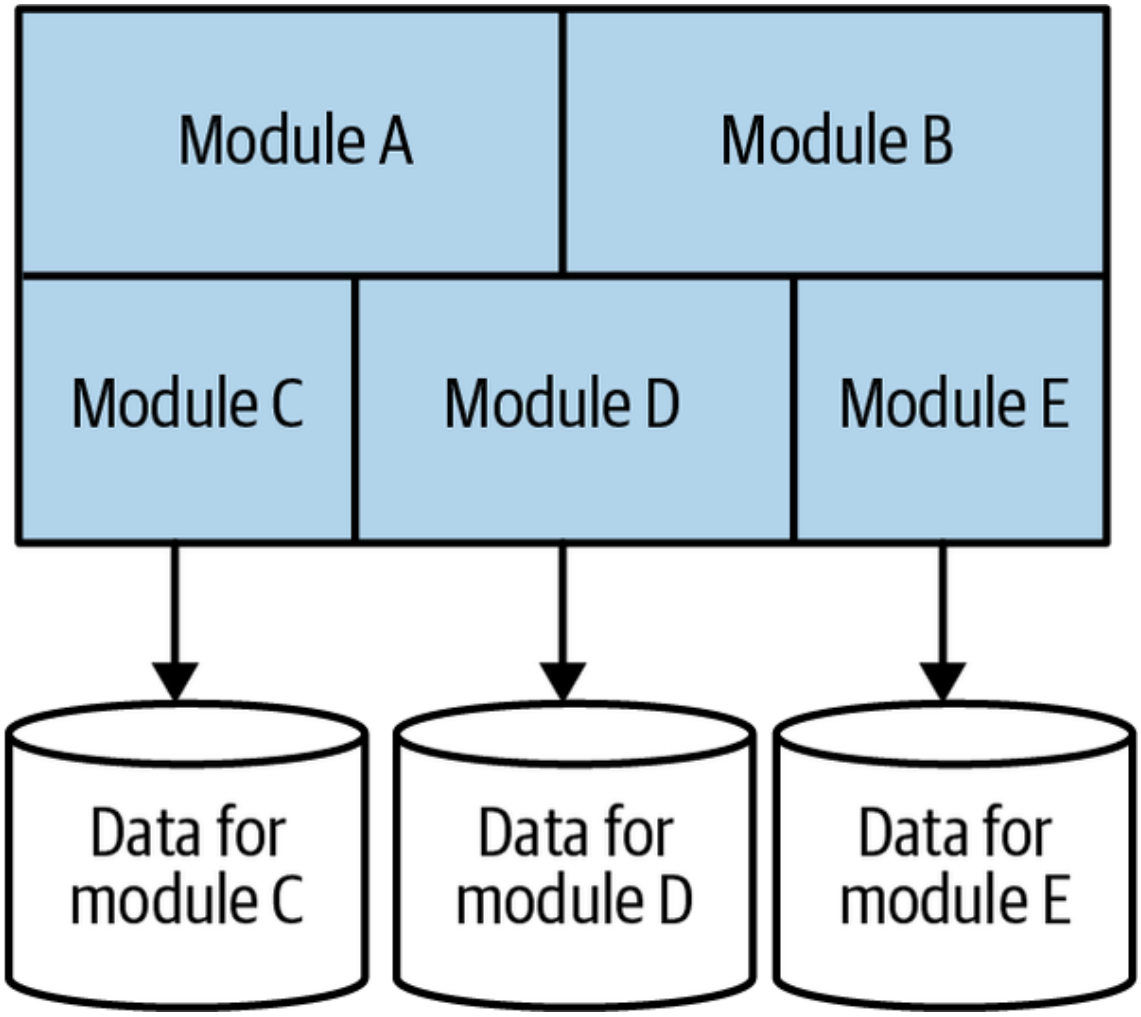

Figure 1-8. A modular monolith with a decomposed database

The Distributed Monolith

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.[7]

Leslie Lamport

A *distributed monolith* is a system that consists of multiple services, but for whatever reason, the entire system must be deployed together. A distributed monolith might well meet the definition of an SOA, but all too often, it fails to deliver on the promises of SOA. In my experience, a distributed monolith has all the disadvantages of a distributed system, *and* the disadvantages of a single-process monolith, without having enough of the upsides of either. Encountering a number of distributed monoliths in my work has in large part influenced my own interest in microservice architecture.

Distributed monoliths typically emerge in an environment in which not enough focus was placed on concepts like information hiding and cohesion of business functionality. Instead, highly coupled architectures cause changes to ripple across service boundaries, and seemingly innocent changes that appear to be local in scope break other parts of the system.

Monoliths and Delivery Contention

As more and more people work in the same place, they get in one another's way—for example, different developers wanting to change the same piece of code, different teams wanting to push functionality live at different times (or to delay deployments), and confusion around who owns what and who makes decisions. A multitude of studies have shown the challenges of confused lines of ownership.[8] I refer to this problem as *delivery contention*.

Having a monolith doesn't mean you will definitely face the challenges of delivery contention any more than having a microservice architecture means that you won't ever face the problem. But a microservice architecture does give you more concrete boundaries around which ownership lines can be drawn in a system, giving you much more flexibility when it comes to reducing this problem.

Advantages of Monoliths

Some monoliths, such as the single-process or modular monoliths, have a whole host of advantages too. Their much simpler deployment topology can avoid many of the pitfalls associated with distributed systems. This can result in much simpler developer workflows, and monitoring, troubleshooting, and activities like end-to-end testing can be greatly simplified as well.

Monoliths can also simplify code reuse within the monolith itself. If we want to reuse code within a distributed system, we need to decide whether we want to copy code, break out libraries, or push the shared functionality into a service. With a monolith, our choices are much simpler, and many people like that simplicity—all the code is there; just use it!

Unfortunately, people have come to view the monolith as something to be avoided—as something inherently problematic. I've met multiple people for whom the term *monolith* is synonymous with *legacy*. This is a problem. A monolithic architecture is a choice, and a valid one at that. I'd go further and say that in my opinion it is the sensible default choice as an architectural style. In other words, I am looking for a reason to be convinced to use microservices, rather than looking for a reason not to use them.

If we fall into the trap of systematically undermining the monolith as a viable option for delivering our software, we're at risk of not doing right by ourselves or by the users of our software.

Enabling Technology

As I touched on earlier, I don't think you need to adopt lots of new technology when you first start using microservices. In fact, that can be counterproductive. Instead, as you ramp up your microservice architecture, you should constantly be looking for issues caused by your increasingly distributed system, and then for technology that might help.

That said, technology has played a large part in the adoption of microservices as a concept. Understanding the tools that are available to help you get the most out of this architecture is going to be a key part of making any implementation of microservices a success. In fact, I would go as far to say that microservices require an understanding of the supporting technology to such a degree that previous distinctions between logical and physical architecture can be problematic—if you are involved in helping shape a microservice architecture, you'll need a breadth of understanding of these two worlds.

We'll be exploring a lot of this technology in detail in subsequent chapters, but before that, let's briefly introduce some of the enabling technology that might help you if you decide to make use of microservices.

Log Aggregation and Distributed Tracing

With the increasing number of processes you are managing, it can be difficult to understand how your system is behaving in a production setting. This can in turn make troubleshooting much more difficult. We'll be exploring these ideas in more depth in Chapter 10, but at a bare minimum, I strongly advocate the implementation of a log aggregation system as a prerequisite for adopting a microservice architecture.

Tip

Be cautious about taking on too much new technology when you start off with microservices. That said, a log aggregation tool is so essential that you should consider it a prerequisite for adopting microservices.

These systems allow you to collect and aggregate logs from across all your services, providing you a central place from which logs can be analyzed, and even made part of an active alerting mechanism. Many options in this space cater to numerous situations. I'm a big fan of Humio for several reasons, but the simple logging services provided by the main public cloud vendors might be good enough to get you started.

You can make these log aggregation tools even more useful by implementing correlation IDs, in which a single ID is used for a related set of service calls—for example, the chain of calls that might be triggered due to user interaction. By logging this ID as part of each log entry, isolating the logs associated with a given flow of calls becomes much easier, which in turn makes troubleshooting much easier.

As your system grows in complexity, it becomes essential to consider tools that allow you to better explore what your system is doing, providing the ability to analyze traces across multiple services, detect bottlenecks, and ask questions of your system that you didn't know you would want to ask in the first place. Open source tools can provide some of these features. One example is Jaeger, which focuses on the distributed tracing side of the equation.

But products like Lightstep and Honeycomb (shown in Figure 1-9) take these ideas further. They represent a new generation of tools that move beyond traditional monitoring approaches, making it much easier to explore the state of your running system. You might already have more conventional tools in place, but you really should look at the capabilities these products provide. They've been built from the ground up to solve the sorts of problems that operators of microservice architectures have to deal with.

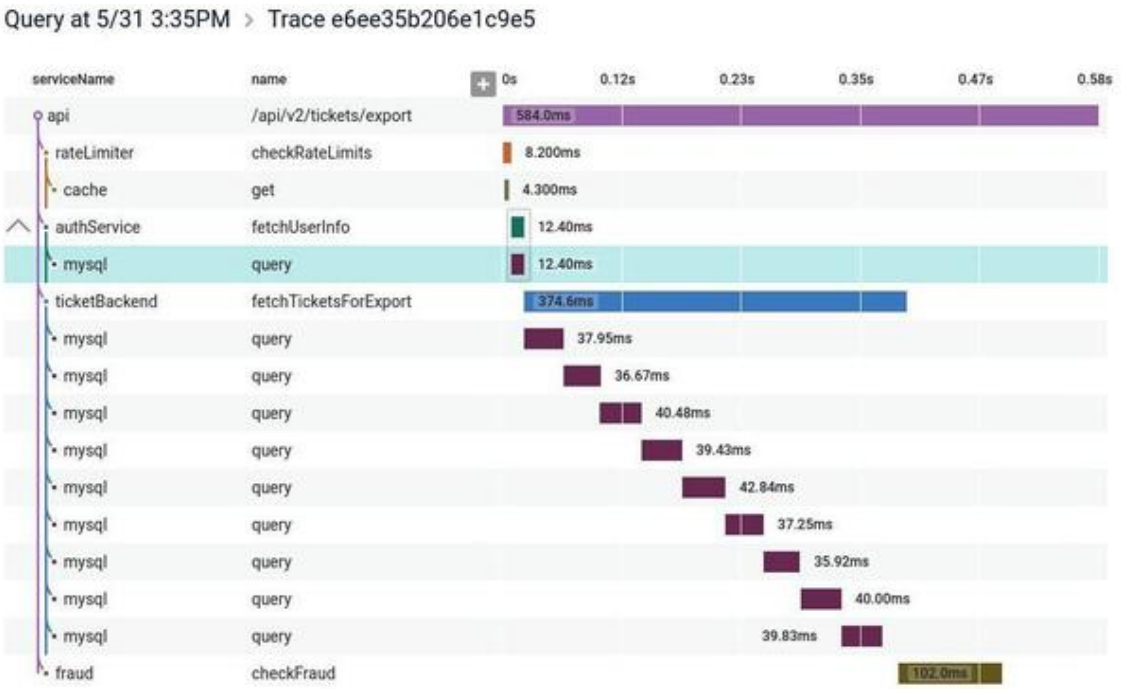

Figure 1-9. A distributed trace shown in Honeycomb, allowing you to identify where time is being spent for operations that can span multiple microservices

Containers and Kubernetes

Ideally, you want to run each microservice instance in isolation. This ensures that issues in one microservice can't affect another microservice—for example, by gobbling up all the CPU. Virtualization is one way to create isolated execution environments on existing hardware, but normal virtualization techniques can be quite heavy when we consider the size of our microservices. *Containers*, on the other hand, provide a much more lightweight way to provision isolated execution for service instances, resulting in faster spin-up times for new container instances, along with being much more cost effective for many architectures.

After you begin playing around with containers, you'll also realize that you need something to allow you to manage these containers across lots of underlying machines. Container orchestration platforms like Kubernetes do exactly that, allowing you to distribute container instances in such a way as to provide the robustness and throughput your service needs, while allowing you to make efficient use of the underlying machines. In Chapter 8 we'll explore the concepts of operational isolation, containers, and Kubernetes.

Don't feel the need to rush to adopt Kubernetes, or even containers for that matter. They absolutely offer significant advantages over more traditional deployment techniques, but their adoption is difficult to justify if you have only a few microservices. After the overhead of managing deployment begins to become a significant headache, start considering containerization of your service and the use of Kubernetes. But if you do end up doing that, do your best to ensure that someone else is running the Kubernetes cluster for you, perhaps by making use of a managed service on a public cloud provider. Running your own Kubernetes cluster can be a significant amount of work!

Streaming

Although with microservices we are moving away from monolithic databases, we still need to find ways to share data between microservices. This is happening at the same time that organizations are wanting to move away from batch reporting operations and toward more real-time feedback, allowing them to react more quickly. Products that allow for the easy streaming and processing of what can often be large volumes of data have therefore become popular with people using microservice architectures.

For many people, Apache Kafka has become the de facto choice for streaming data in a microservice environment, and for good reason. Capabilities such as message permanence, compaction, and the ability to scale to handle large volumes of messages can be incredibly useful. Kafka has started adding stream-processing capabilities in the form of KSQLDB, but you can also use it with dedicated stream-processing solutions like Apache Flink. Debezium is an open source tool developed to help stream data from existing datasources over Kafka, helping ensure that traditional datasources can become part of a stream-based architecture. In Chapter 4 we'll look at how streaming technology can play a part in microservice integration.

Public Cloud and Serverless

Public cloud providers, or more specifically the main three providers—Google Cloud, Microsoft Azure, and Amazon Web Services (AWS)—offer a huge array of managed services and deployment options for managing your application. As your microservice architecture grows, more and more work will be pushed into the operational space. Public cloud providers offer a host of managed services, from managed database instances or Kubernetes clusters to message brokers or distributed filesystems. By making use of these managed services, you are offloading a large amount of this work to a third party that is arguably better able to deal with these tasks.

Of particular interest among the public cloud offerings are the products that sit under the banner of *serverless*. These products hide the underlying machines, allowing you to work at a higher level of abstraction. Examples of serverless products include message brokers, storage solutions, and databases. Function as a Service (FaaS) platforms are of special interest because they provide a nice abstraction around the deployment of code. Rather than worrying about how many servers you need to run your service, you just deploy your code and let the underlying platform handle spinning up instances of your code on demand. We'll look at serverless in more detail in Chapter 8.

Advantages of Microservices

The advantages of microservices are many and varied. Many of these benefits can be laid at the door of any distributed system. Microservices, however, tend to achieve these benefits to a greater degree primarily because they take a more opinionated stance in the way service boundaries are defined. By combining the concepts of information hiding and domain-driven design with the power of distributed systems, microservices can help deliver significant gains over other forms of distributed architectures.

Technology Heterogeneity

With a system composed of multiple, collaborating microservices, we can decide to use different technologies inside each one. This allows us to pick the right tool for each job

rather than having to select a more standardized, one-size-fits-all approach that often ends up being the lowest common denominator.

If one part of our system needs to improve its performance, we might decide to use a different technology stack that is better able to achieve the required performance levels. We might also decide that the way we store our data needs to change for different parts of our system. For example, for a social network, we might store our users' interactions in a graph-oriented database to reflect the highly interconnected nature of a social graph, but perhaps the posts the users make could be stored in a document-oriented data store, giving rise to a heterogeneous architecture like the one shown in Figure 1-10.

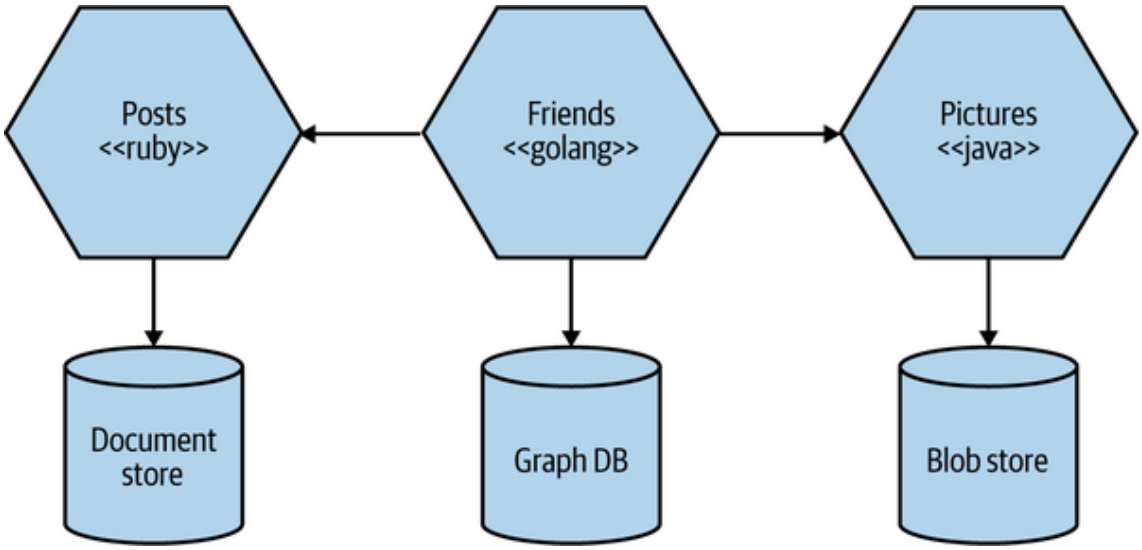

Figure 1-10. Microservices can allow you to more easily embrace different technologies

With microservices, we are also able to more quickly adopt technologies and to understand how new advancements might help us. One of the biggest barriers to trying out and adopting a new technology is the risks associated with it. With a monolithic application, if I want to try a new programming language, database, or framework, any change will affect much of my system. With a system consisting of multiple services, I have multiple new places to try out a new piece of technology. I can pick a microservice with perhaps the lowest risk and use the technology there, knowing that I can limit any potential negative impact. Many organizations find this ability to more quickly absorb new technologies to be a real advantage.

Embracing multiple technologies doesn't come without overhead, of course. Some organizations choose to place some constraints on language choices. Netflix and Twitter, for example, mostly use the Java Virtual Machine (JVM) as a platform because those companies have a very good understanding of the reliability and performance of that system. They also develop libraries and tooling for the JVM that make operating at scale much easier, but the reliance on JVM-specific libraries makes things more difficult for non-Java-based services or clients. But neither Twitter nor Netflix uses only one technology stack for all jobs.

The fact that internal technology implementation is hidden from consumers can also make upgrading technologies easier. Your entire microservice architecture might be based on Spring Boot, for example, but you could change JVM version or framework versions for just one microservice, making it easier to manage the risk of upgrades.

Robustness

A key concept in improving the robustness of your application is the bulkhead. A component of a system may fail, but as long as that failure doesn't cascade, you can isolate the problem, and the rest of the system can carry on working.
Service boundaries become your obvious bulkheads. In a monolithic service, if the service fails, everything stops working. With a monolithic system, we can run on multiple machines to reduce our chance of failure, but with microservices, we can build systems that handle the total failure of some of the constituent services and degrade functionality accordingly.

We do need to be careful, however. To ensure that our microservice systems can properly embrace this improved robustness, we need to understand the new sources of failure that distributed systems have to deal with. Networks can and will fail, as will machines. We need to know how to handle such failures and the impact (if any) those failures will have on the end users of our software. I have certainly worked with teams who have ended up with a less robust system after their migration to microservices due to their not taking these concerns seriously enough.

Scaling

With a large, monolithic service, we need to scale everything together. Perhaps one small part of our overall system is constrained in performance, but if that behavior is locked up in a giant monolithic application, we need to handle scaling everything as a piece. With smaller services, we can scale just those services that need scaling, allowing us to run other parts of the system on smaller, less powerful hardware, as illustrated in Figure 1-11.

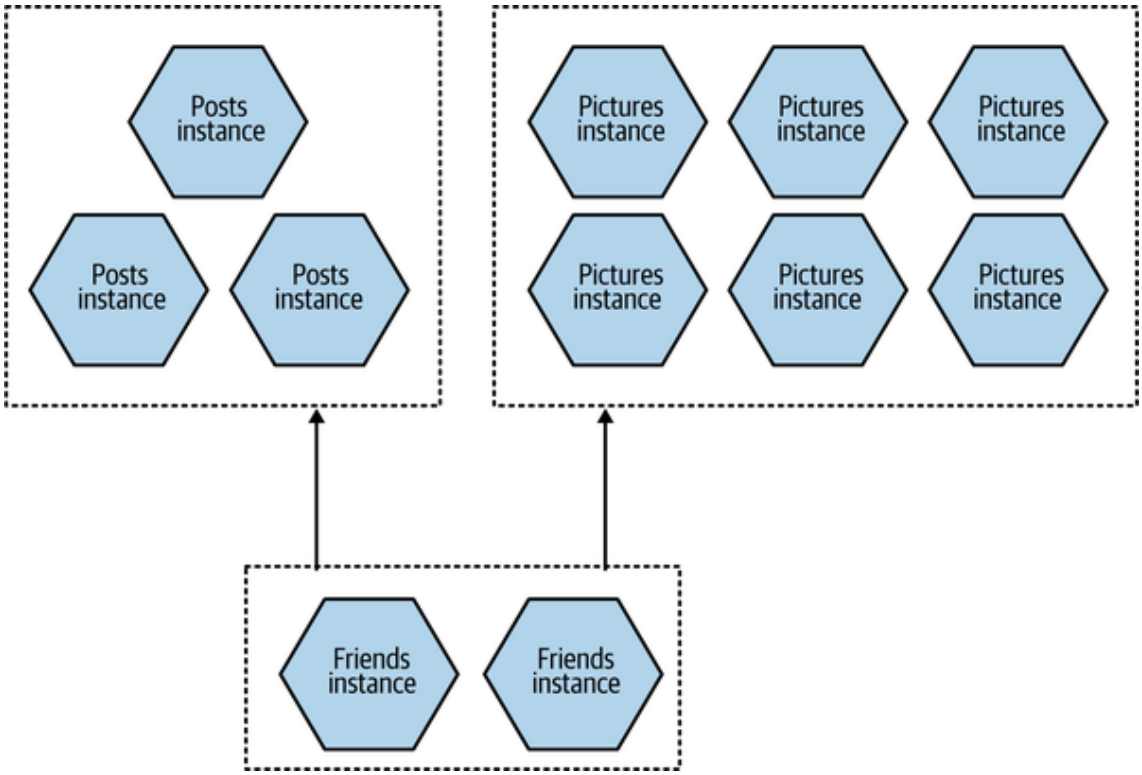

Figure 1-11. You can target scaling at just the microservices that need it

Gilt, an online fashion retailer, adopted microservices for this exact reason. Having started in 2007 with a monolithic Rails application, by 2009 Gilt's system was unable to cope with the load being placed on it. By splitting out core parts of its system, Gilt was better able to deal with its traffic spikes, and today it has more than 450 microservices, each one running on multiple separate machines.

When embracing on-demand provisioning systems like those provided by AWS, we can even apply this scaling on demand for those pieces that need it. This allows us to control our costs more effectively. It's not often that an architectural approach can be so closely correlated to an almost immediate cost savings.

Ultimately, we can scale our applications in a multitude of ways, and microservices can be an effective part of this. We'll look at the scaling of microservices in more detail in Chapter 13.

Ease of Deployment

A one-line change to a million-line monolithic application requires the entire application to be deployed in order to release the change. That could be a large-impact, high-risk deployment. In practice, deployments such as these end up happening infrequently because of understandable fear. Unfortunately, this means that our changes continue to build up between releases, until the new version of our application entering production has masses of changes. And the bigger the delta between releases, the higher the risk that we'll get something wrong!

With microservices, we can make a change to a single service and deploy it independently of the rest of the system. This allows us to get our code deployed more quickly. If a problem does occur, it can be quickly isolated to an individual service, making fast rollback easy to achieve. It also means that we can get our new functionality out to customers more quickly. This is one of the main reasons organizations like Amazon and Netflix use these architectures—to ensure that they remove as many impediments as possible to getting software out the door.

Organizational Alignment

Many of us have experienced the problems associated with large teams and large codebases. These problems can be exacerbated when the team is distributed. We also know that smaller teams working on smaller codebases tend to be more productive.

Microservices allow us to better align our architecture to our organization, helping us minimize the number of people working on any one codebase to hit the sweet spot of team size and productivity. Microservices also allow us to change ownership of services as the organization changes—enabling us to maintain the alignment between architecture and organization in the future.

Composability

One of the key promises of distributed systems and service-oriented architectures is that we open up opportunities for reuse of functionality. With microservices, we allow for our functionality to be consumed in different ways for different purposes. This can be especially important when we think about how our consumers use our software.

Gone is the time when we could think narrowly about either our desktop website or our mobile application. Now we need to think of the myriad ways that we might want to weave together capabilities for the web, native application, mobile web, tablet app, or wearable device. As organizations move away from thinking in terms of narrow channels to embracing more holistic concepts of customer engagement, we need architectures that can keep up.

With microservices, think of us opening up seams in our system that are addressable by outside parties. As circumstances change, we can build applications in different ways. With a monolithic application, I often have one coarse-grained seam that can be used from the outside. If I want to break that up to get something more useful, I'll need a hammer!

Microservice Pain Points

Microservice architectures bring a host of benefits, as we've already seen. But they also bring a host of complexity. If you are considering adopting a microservice architecture, it's important that you be able to compare the good with the bad. In reality, most microservice points can be laid at the door of distributed systems and thus would just as likely be evident in a distributed monolith as in a microservice architecture.

We'll be covering many of these issues in depth throughout the rest of the book—in fact, I'd argue that the bulk of this book is about dealing with the pain, suffering, and horror of owning a microservice architecture.

Developer Experience

As you have more and more services, the developer experience can begin to suffer. More resource-intensive runtimes like the JVM can limit the number of microservices that can be run on a single developer machine. I could probably run four or five JVM-based microservices as separate processes on my laptop, but could I run 10 or 20? Most likely not. Even with less taxing runtimes, there is a limit to the number of things you can run locally, which inevitably will start conversations about what to do when you can't run the entire system on one machine. This can become even more complicated if you are using cloud services that you cannot run locally.

Extreme solutions can involve "developing in the cloud," where developers move away from being able to develop locally anymore. I'm not a fan of this, because feedback cycles can suffer greatly. Instead, I think limiting the scope of which parts of a system a developer needs to work on is likely to be a much more straightforward approach. However, this might be problematic if you want to embrace more of a "collective ownership" model in which any developer is expected to work on any part of the system.

Technology Overload

The sheer weight of new technology that has sprung up to enable the adoption of microservice architectures can be overwhelming. I'll be honest and say that a lot of this technology has just been rebranded as "microservice friendly," but some advances have legitimately helped in dealing with the complexity of these sorts of architectures. There is a danger, though, that this wealth of new toys can lead to a form of technology fetishism. I've seen so many companies adopting microservice architecture who decided that it was also the best time to introduce vast arrays of new and often alien technology.

Microservices may well give you the *option* for each microservice to be written in a different programming language, to run on a different runtime, or to use a different database—but these are options, not requirements. You have to carefully balance the breadth and complexity of the technology you use against the costs that a diverse array of technology can bring.

When you start adopting microservices, some fundamental challenges are inescapable: you'll need to spend a lot of time understanding issues around data consistency, latency, service modeling, and the like. If you're trying to understand how these ideas change the way you think about software development at the same time that you're embracing a huge amount of new technology, you'll have a hard time of it. It's also worth pointing out that the bandwidth taken up by trying to understand all of this new technology will reduce the time you have for actually shipping features to your users.

As you (gradually) increase the complexity of your microservice architecture, look to introduce new technology as you need it. You don't need a Kubernetes cluster when you have three services! In addition to ensuring that you're not overloaded with the complexity of these new tools, this gradual increase has the added benefit of allowing you to gain new and better ways of doing things that will no doubt emerge over time.

Cost

It's highly likely that in the short term at least you'll see an increase in costs from a number of factors. Firstly, you'll likely need to run more things—more processes, more computers, more network, more storage, and more supporting software (which will incur additional license fees).

Secondly, any change you introduce into a team or an organization will slow you down in the short term. It takes time to learn new ideas, and to work out how to use them effectively. While this is going on, other activities will be impacted. This will result in either a direct slowdown in delivery of new functionality or the need to add more people to offset this cost.

In my experience, microservices are a poor choice for an organization primarily concerned with reducing costs, as a cost-cutting mentality—where IT is seen as a cost center rather than a profit center—will constantly be a drag on getting the most out of this architecture. On the other hand, microservices can help you make more money if you can use these architectures to reach more customers or develop more functionality in parallel. So are microservices a way to drive more profits? Perhaps. Are microservices a way to reduce costs? Not so much.

Reporting

With a monolithic system, you typically have a monolithic database. This means that stakeholders who want to analyze all the data together, often involving large join operations across data, have a ready-made schema against which to run their reports. They can just run them directly against the monolithic database, perhaps against a read replica, as shown in Figure 1-12.

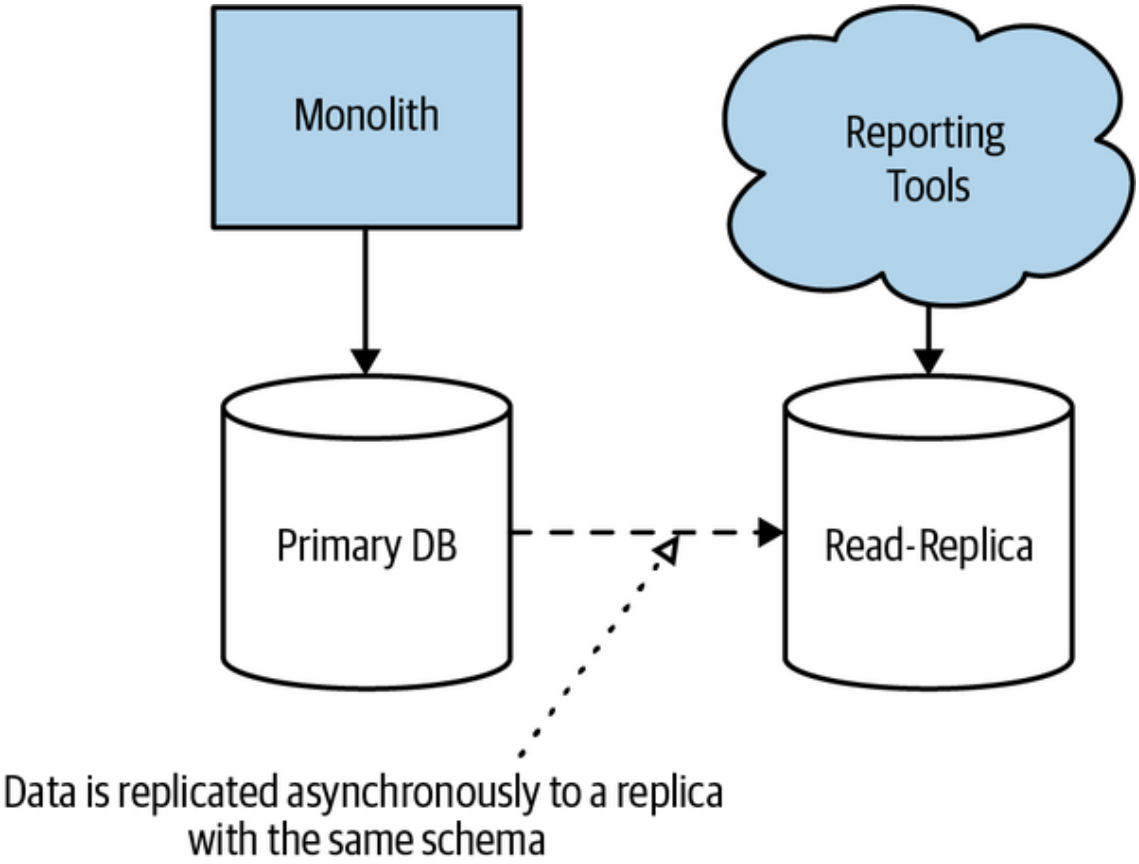

Figure 1-12. Reporting carried out directly on the database of a monolith

With a microservice architecture, we have broken up this monolithic schema. That doesn't mean that the need for reporting across all our data has gone away; we've just made it much more difficult, because now our data is scattered across multiple logically isolated schemas.

More modern approaches to reporting, such as using streaming to allow for real-time reporting on large volumes of data, can work well with a microservice architecture but typically require the adoption of new ideas and associated technology. Alternatively, you might simply need to publish data from your microservices into central reporting databases (or perhaps less structured data lakes) to allow for reporting use cases.

Monitoring and Troubleshooting

With a standard monolithic application, we can have a fairly simplistic approach to monitoring. We have a small number of machines to worry about, and the failure mode of the application is somewhat binary—the application is often either all up or all down. With a microservice architecture, do we understand the impact if just a single instance of a service goes down?

With a monolithic system, if our CPU is stuck at 100% for a long time, we know it's a big problem. With a microservice architecture with tens or hundreds of processes, can we say the same thing? Do we need to wake someone up at 3 a.m. when just one process is stuck at 100% CPU?

Luckily, there are a whole host of ideas in this space that can help. If you'd like to explore this concept in more detail, I recommend *Distributed Systems Observability* by Cindy

Sridharan (O'Reilly) as an excellent starting point, although we'll also be taking our own look at monitoring and observability in Chapter 10.

Security

With a single-process monolithic system, much of our information flowed within that process. Now, more information flows over networks between our services. This can make our data more vulnerable to being observed in transit and also to potentially being manipulated as part of man-in-the-middle attacks. This means that you might need to direct more care to protecting data in transit and to ensuring that your microservice endpoints are protected so that only authorized parties are able to make use of them. Chapter 11 is dedicated entirely to looking at the challenges in this space.

Testing

With any type of automated functional test, you have a delicate balancing act. The more functionality a test executes—i.e., the broader the scope of the test—the more confidence you have in your application. On the other hand, the larger the scope of the test, the harder it is to set up test data and supporting fixtures, the longer the test can take to run, and the harder it can be to work out what is broken when it fails. In Chapter 9 I'll share a number of techniques for making testing work in this more challenging environment.

End-to-end tests for any type of system are at the extreme end of the scale in terms of the functionality they cover, and we are used to them being more problematic to write and maintain than smaller-scoped unit tests. Often this is worth it, though, because we want the confidence that comes from having an end-to-end test use our systems in the same way a user might.

But with a microservice architecture, the scope of our end-to-end tests becomes *very* large. We would now need to run tests across multiple processes, all of which need to be deployed and appropriately configured for the test scenarios. We also need to be prepared for the false negatives that occur when environmental issues, such as service instances dying or network time-outs of failed deployments, cause our tests to fail.

These forces mean that as your microservice architecture grows, you will get a diminishing return on investment when it comes to end-to-end testing. The testing will cost more but won't manage to give you the same level of confidence that it did in the past. This will drive you toward new forms of testing, such as contract-driven testing or testing in production, as well as the exploration of progressive delivery techniques such as parallel runs or canary releases, which we'll look at in Chapter 8.

Latency

With a microservice architecture, processing that might previously have been done locally on one processor can now end up being split across multiple separate microservices. Information that previously flowed within only a single process now needs to be serialized, transmitted, and deserialized over networks that you might be exercising more than ever before. All of this can result in worsening latency of your system.

Although it can be difficult to measure the exact impact on latency of operations at the design or coding phase, this is another reason it's important to undertake any microservice migration in an incremental fashion. Make a small change and then measure

the impact. This assumes that you have some way of measuring the end-to-end latency for the operations you care about—distributed tracing tools like Jaeger can help here. But you also need to have an understanding of what latency is acceptable for these operations. Sometimes making an operation slower is perfectly acceptable, as long as it is still fast enough!

Data Consistency

Shifting from a monolithic system, in which data is stored and managed in a single database, to a much more distributed system, in which multiple processes manage state in different databases, causes potential challenges with respect to consistency of data. Whereas in the past you might have relied on database transactions to manage state changes, you'll need to understand that similar safety cannot easily be provided in a distributed system. The use of distributed transactions in most cases proves to be highly problematic in coordinating state changes.

Instead, you might need to start using concepts like sagas (something I'll detail at length in Chapter 6) and eventual consistency to manage and reason about state in your system. These ideas can require fundamental changes in the way you think about data in your systems, something that can be quite daunting when migrating existing systems. Yet again, this is another good reason to be cautious in how quickly you decompose your application. Adopting an incremental approach to decomposition, so that you are able to assess the impact of changes to your architecture in production, is really important.

Should I Use Microservices?

Despite the drive in some quarters to make microservice architectures the default approach for software, I feel that because of the numerous challenges I've outlined, adopting them still requires careful thought. You need to assess your own problem space, skills, and technology landscape and understand what you are trying to achieve before deciding whether microservices are right for you. They are *an* architectural approach, not *the* architectural approach. Your own context should play a huge part in your decision whether to go down that path.

That said, I want to outline a few situations that would typically tip me away from—or toward—picking microservices.

Whom They Might Not Work For

Given the importance of defining stable service boundaries, I feel that microservice architectures are often a bad choice for brand-new products or startups. In either case, the domain that you are working with is typically undergoing significant change as you iterate on the fundamentals of what you are trying to build. This shift in domain models will, in turn, result in more changes being made to service boundaries, and coordinating changes across service boundaries is an expensive undertaking. In general, I feel it's more appropriate to wait until enough of the domain model has stabilized before looking to define service boundaries.

I do see a temptation for startups to go microservice first, the reasoning being, "If we're really successful, we'll need to scale!" The problem is that you don't necessarily know if anyone is even going to want to use your new product. And even if you do become successful enough to require a highly scalable architecture, the thing you end up

delivering to your users might be very different from what you started building in the first place. Uber initially focused on limos, and Flickr spun out of attempts to create a multiplayer online game. The process of finding product market fit means that you might end up with a very different product at the end than the one you thought you'd build when you started.

Startups also typically have fewer people available to build the system, which creates more challenges with respect to microservices. Microservices bring with them sources of new work and complexity, and this can tie up valuable bandwidth. The smaller the team, the more pronounced this cost will be. When working with smaller teams with just a handful of developers, I'm very hesitant to suggest microservices for this reason.

The challenge of microservices for startups is compounded by the fact that normally your biggest constraint is people. For a small team, a microservice architecture can be difficult to justify because there is work required just to handle the deployment and management of the microservices themselves. Some people have described this as the "microservice tax." When that investment benefits lots of people, it's easier to justify. But if one person out of your five-person team is spending their time on these issues, that's a lot of valuable time not being spent building your product. It's much easier to move to microservices later, after you understand where the constraints are in your architecture and what your pain points are—then you can focus your energy on using microservices in the most sensible places.

Finally, organizations creating software that will be deployed and managed by their customers may struggle with microservices. As we've already covered, microservice architectures can push a lot of complexity into the deployment and operational domain. If you are running the software yourself, you are able to offset this new complexity by adopting new technology, developing new skills, and changing working practices. This isn't something you can expect your customers to do. If they are used to receiving your software as a Windows installer, it's going to come as an awful shock to them when you send out the next version of your software and say, "Just put these 20 pods on your Kubernetes cluster!" In all likelihood, they will have no idea what a pod, Kubernetes, or a cluster even is.

Where They Work Well

In my experience, probably the single biggest reason that organizations adopt microservices is to allow for more developers to work on the same system without getting in each other's way. Get your architecture and organizational boundaries right, and you allow more people to work independently of each other, reducing delivery contention. A five-person startup is likely to find a microservice architecture a drag. A hundred-person scale-up that is growing rapidly is likely to find that its growth is much easier to accommodate with a microservice architecture properly aligned around its product development efforts.

Software as a Service (SaaS) applications are, in general, also a good fit for a microservice architecture. These products are typically expected to operate 24-7, which creates challenges when it comes to rolling out changes. The independent releasability of microservice architectures is a huge boon in this area. Furthermore, the microservices can be scaled up or down as required. This means that as you establish a sensible

baseline for your system's load characteristics, you get more control over ensuring that you can scale your system in the most cost-effective way possible.

The technology-agnostic nature of microservices ensures that you can get the most out of cloud platforms. Public cloud vendors provide a wide array of services and deployment mechanisms for your code. You can much more easily match the requirements of specific services to the cloud services that will best help you implement them. For example, you might decide to deploy one service as a set of functions, another as a managed virtual machine (VM), and another on a managed Platform as a Service (PaaS) platform.

Although it's worth noting that adopting a wide range of technology can often be a problem, being able to try out new technology easily is a good way to rapidly identify new approaches that might yield benefits. The growing popularity of FaaS platforms is one such example. For the appropriate workloads, an FaaS platform can drastically reduce the amount of operational overhead, but at present, it's not a deployment mechanism that would be suitable in all cases.

Microservices also present clear benefits for organizations looking to provide services to their customers over a variety of new channels. A lot of digital transformation efforts seem to involve trying to unlock functionality hidden away in existing systems. The desire is to create new customer experiences that can support the needs of users via whatever interaction mechanism makes the most sense.

Above all, a microservice architecture is one that can give you a lot of flexibility as you continue to evolve your system. That flexibility has a cost, of course, but if you want to keep your options open regarding changes you might want to make in the future, it could be a price worth paying.

Summary

Microservice architectures can give you a huge degree of flexibility in choosing technology, handling robustness and scaling, organizing teams, and more. This flexibility is in part why many people are embracing microservice architectures. But microservices bring with them a significant degree of complexity, and you need to ensure that this complexity is warranted. For many, they have become a default system architecture, to be used in virtually all situations. However, I still think that they are an architectural choice whose use must be justified by the problems you are trying to solve; often, simpler approaches can deliver much more easily.

Nonetheless, many organizations, especially larger ones, have shown how effective microservices can be. When the core concepts of microservices are properly understood and implemented, they can help create empowering, productive architectures that can help systems become more than the sum of their parts.

I hope this chapter has served as a good introduction to these topics. Next, we're going to look at how we define microservice boundaries, exploring the topics of structured programming and domain-driven design along the way.

1 This concept was first outlined by David Parnas in "Information Distribution Aspects of Design Methodology", *Information Processing: Proceedings of the IFIP Congress 1971* (Amsterdam: North-Holland, 1972), 1:339–44.

2 Alistair Cockburn, “Hexagonal Architecture,” January 4, 2005, *https://oreil.ly/NfvTP*.

3 For an in-depth introduction to domain-driven design, see *Domain-Driven Design* by Eric Evans (Addison-Wesley)—or for a more condensed overview, see *Domain-Driven Design Distilled* by Vaughn Vernon (Addison-Wesley).

4 Matthew Skelton and Manuel Pais, *Team Topologies* (Portland, OR: IT Revolution, 2019).

5 David Heinemeier Hansson, “The Majestic Monolith,” Signal v. Noise, February 29, 2016, *https://oreil.ly/WwG1C*.

6 For some useful insights into the thinking behind Shopify’s use of a modular monolith rather than microservices, watch “Deconstructing the Monolith” by Kirsten Westeinde.

7 Leslie Lamport, email message to a DEC SRC bulletin board at 12:23:29 PDT on May 28, 1987.

8 Microsoft Research has carried out studies in this space, and I recommend all of them, but as a starting point, I suggest “Don’t Touch My Code! Examining the Effects of Ownership on Software Quality” by Christian Bird et al.