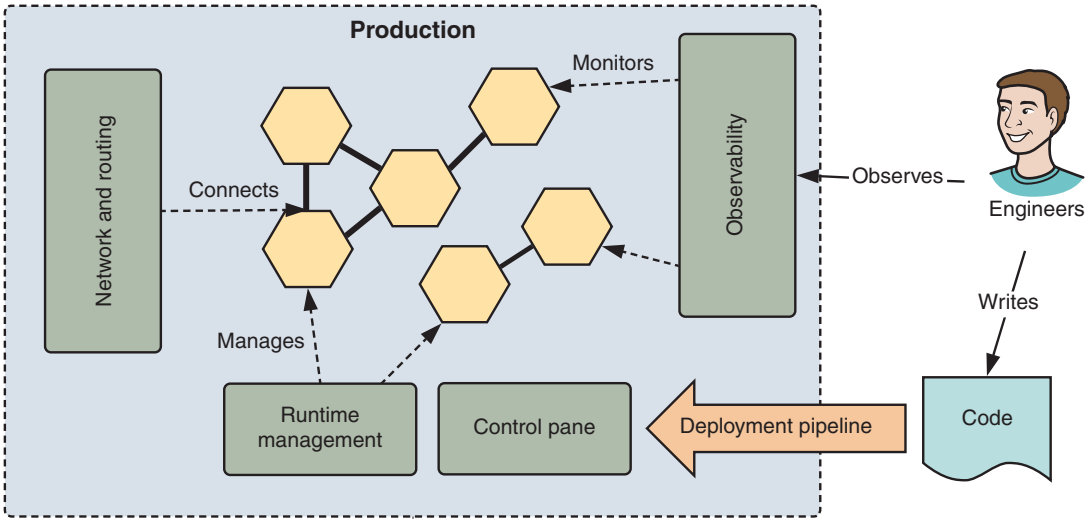
Microservices IN ACTION

Morgan Bruce
Paulo A. Pereira

Sample Chapter

MANNING

A microservice production environment



A microservice production environment has several components: a deployment target, a deployment pipeline, runtime management, networking features, and support for observability. In this book, we'll teach you about these components and how you can use them to build a stable, modern microservice application.

Microservices in Action

by Morgan Bruce
Paulo A. Pereira

Chapter 1

brief contents

Part 1

The lay of the land

This part introduces microservice architecture, explores the properties and benefits of microservice applications, and presents some of the challenges you'll face in developing microservice applications. We'll also introduce SimpleBank, a fictional company whose attempts to build a microservice application will be the common thread in many examples used in this book.

1 Designing and running microservices

This chapter covers

- Defining a microservice application
- The challenges of a microservices approach
- Approaches to designing a microservice application
- Approaches to running microservices successfully

Software developers strive to craft effective and timely solutions to complex problems. The first problem you usually try to solve is: What does your customer want? If you're skilled (or lucky), you get that right. But your efforts rarely stop there. Your successful application continues to grow: you debug issues; you build new features; you keep it available and running smoothly.

Even the most disciplined teams can struggle to sustain their early pace and agility in the face of a growing application. At worst, your once simple and stable product becomes both intractable and delicate. Instead of sustainably delivering more value to your customers, you're fatigued from outages, anxious about releasing, and too slow to deliver new features or fixes. Neither your customers nor your developers are happy.

Microservices promise a better way to sustainably deliver business impact. Rather than a single monolithic unit, applications built using microservices are made up of

loosely coupled, autonomous services. By building services that do one thing well, you can avoid the inertia and entropy of large applications. Even in existing applications, you can progressively extract functionality into independent services to make your whole system more maintainable.

When we started working with microservices, we quickly realized that building smaller and more self-contained services was only one part of running a stable and business-critical application. After all, any successful application will spend much more of its life in production than in a code editor. To deliver value with microservices, our team couldn't be focused on build alone. We needed to be skilled at operations: deployment, observation, and diagnosis.

1.1 What is a microservice application?

A microservice application is a collection of autonomous services, each of which does one thing well, that work together to perform more intricate operations. Instead of a single complex system, you build and manage a suite of relatively simple services that might interact in complex ways. These services collaborate with each other through technology-agnostic messaging protocols, either point-to-point or asynchronously.

This might seem like a simple idea, but it has striking implications for reducing friction in the development of complex systems. Classical software engineering practice advocates *high cohesion* and *loose coupling* as desirable properties of a well-engineered system. A system that has these properties will be easier to maintain and more malleable in the face of change.

Cohesion is the degree to which elements of a certain module belong together, whereas coupling is the degree to which one element knows about the inner workings of another. Robert C. Martin's Single Responsibility Principle is a useful way to consider the former:

> *Gather together the things that change for the same reasons. Separate those things that change for different reasons.*

In a monolithic application, you try to design for these properties at a class, module, or library level. In a microservice application, you aim instead to attain these properties at the level of independently deployable units of functionality. A single microservice should be highly cohesive: it should be responsible for some single capability within an application. Likewise, the less that each service knows about the inner workings of other services, the easier it is to make changes to one service—or capability—without forcing changes to others.

To get a better picture of how a microservice application fits together, let's start by considering some of the features of an online investment tool:

- Opening an account
- Depositing and withdrawing money
- Placing orders to buy or sell positions in financial products (for example, shares)
- Modeling risk and making financial predictions

Let's explore the process of selling shares:

1. A user creates an order to sell some shares of a stock from their account.
2. This position is reserved on their account, so it can't be sold multiple times.

3 It costs money to place an order on the market—the account is charged a fee.
4 The system needs to communicate that order to the appropriate stock market.

Figure 1.1 shows how placing that sell order might look as part of a microservice application.

You can observe three key characteristics of microservices in figure 1.1:

- Each microservice is *responsible for a single capability. This might be business related or represent a shared technical capability, such as integration with a third party (for example, the stock exchange).*
- A microservice *owns its data store,* if it has one. This reduces coupling between services because other services can only access data they don't own through the interface that a service provides.
- Microservices themselves, not the messaging mechanism that connects them nor another piece of software, are *responsible for choreography and collaboration*—the sequencing of messages and actions to perform some useful activity.

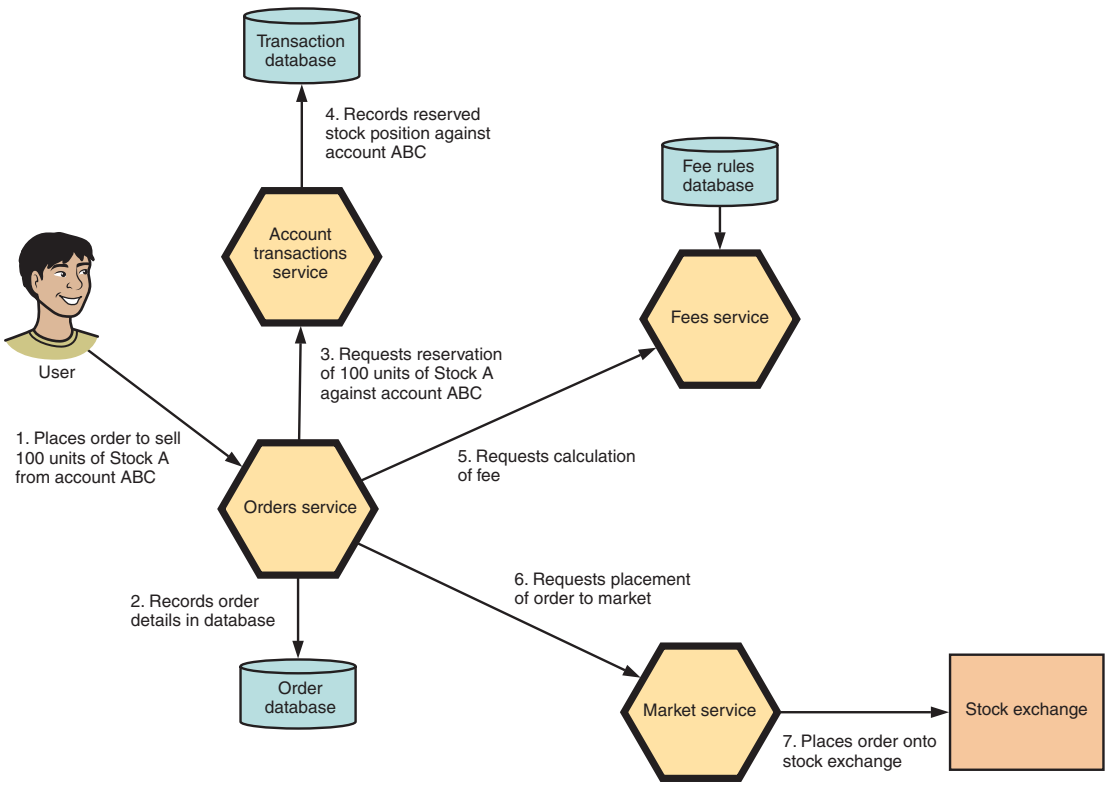


Figure 1.1 The flow of communication through microservices in an application that allows users to sell positions in financial shares

In addition to these three characteristics, you can identify two more fundamental attributes of microservices:

- Each microservice can be *deployed independently*. Without this, a microservice application would still be monolithic at the point of deployment.
- A microservice is *replaceable*. Having a single capability places natural bounds on size; likewise, it makes the individual responsibility, or role, of a service easy to comprehend.

The idea that microservices are responsible for coordinating actions in a system is the crucial difference between this approach and traditional service-oriented architectures (SOAs). Those types of systems often used enterprise service buses (ESBs) or more complex orchestration standards to externalize messaging and process orchestration from applications themselves. In that model, services often lacked cohesion, as business logic was increasingly added to the service bus, rather than the services themselves.

It's interesting to think about how decoupling functionality in the online investment system helps you be more flexible in the face of changing requirements. Imagine that you need to change how fees are calculated. You could make and release those changes to the *fees* service without any change to its upstream or downstream services. Or imagine an entirely new requirement: when an order is placed, you need to alert your risk team if it doesn't match normal trading patterns. It'd be easy to build a new microservice to perform that operation based on an event raised by the *orders* service without changing the rest of the system.

1.1.1 Scaling through decomposition

You also can consider how microservices allow you to scale an application. In *The Art of Scalability*, Abbott and Fisher define three dimensions of scale as the scale cube (figure 1.2).

Monolithic applications typically scale through horizontal duplication: deploying multiple, identical instances of the application. This is also known as cookie-cutter, or X-axis, scaling. Conversely, microservice applications are an example of Y-axis scaling, where you decompose a system to address the unique scaling needs of different functionality.

> **NOTE** The Z axis refers to horizontal data partitions: sharding. You can apply sharding to either approach — microservices or monolithic applications — but we won't be exploring that topic in this book.

Let's revisit the investment tool as an example, with the following characteristics:

- Financial predictions might be computationally onerous and are rarely done.
- Complex regulatory and business rules may govern investment accounts.
- Market trading may happen in extremely large volumes, while also relying on minimizing latency.

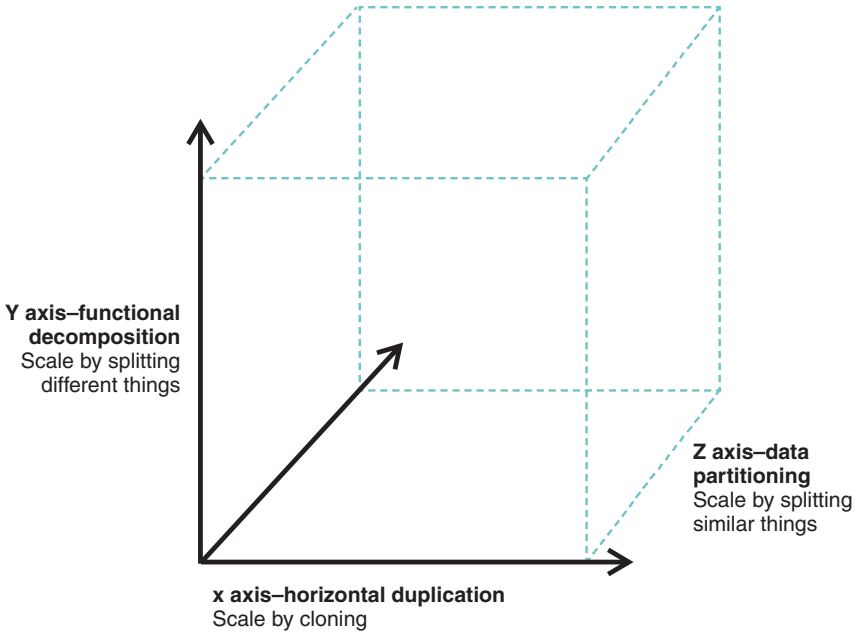


Figure 1.2 The three dimensions of scaling an application

If you build features as microservices that meet the requirements of these characteristics, you can choose the ideal technical tools to solve each problem, rather than trying to fit square pegs into round holes. Likewise, autonomy and independent deployment mean you can manage the microservices' underlying resource needs separately. Interestingly, this also implies a natural way to limit failure: if your financial prediction service fails, that failure is unlikely to cascade to the market trading or investment account services.

Microservice applications have some interesting technical properties:

- Building services along the lines of single capabilities places natural bounds on size and responsibility.
- Autonomy allows you to develop, deploy, and scale services independently.

1.1.2 Key principles

Five cultural and architectural principles underpin microservices development:

- Autonomy
- Resilience
- Transparency
- Automation
- Alignment

These principles should drive your technical and organizational decisions when you're building and running a microservice application. Let's explore each of them.

AUTONOMY

We've established that microservices are *autonomous*—each service operates *and changes independently of others*. To ensure that autonomy, you need to design your services so they are:

- *Loosely coupled*—By interacting through clearly defined interfaces, or through published events, each microservice remains independent of the internal implementation of its collaborators. For example, the orders service we introduced earlier shouldn't be aware of the implementation of the account transactions service. This is illustrated in figure 1.3.
- *Independently deployable*—Services will be developed in parallel, often by multiple teams. Being forced to deploy them in lockstep or in an orchestrated formation would result in risky and anxious deployments. Ideally, you want to use your smaller services to enable rapid, frequent, and small releases.

Autonomy is also cultural. It's vital that you delegate accountability for and ownership of services to teams responsible for delivering business impact. As we've established, organizational design has an influence on system design. Clear service ownership allows teams to build iteratively and make decisions based on their local context and goals. Likewise, this model is ideal for promoting end-to-end ownership, where a team is responsible for a service in both development and production.

NOTE In chapter 13, we'll discuss developing responsible and autonomous engineering teams and why this is crucial when working with microservices.

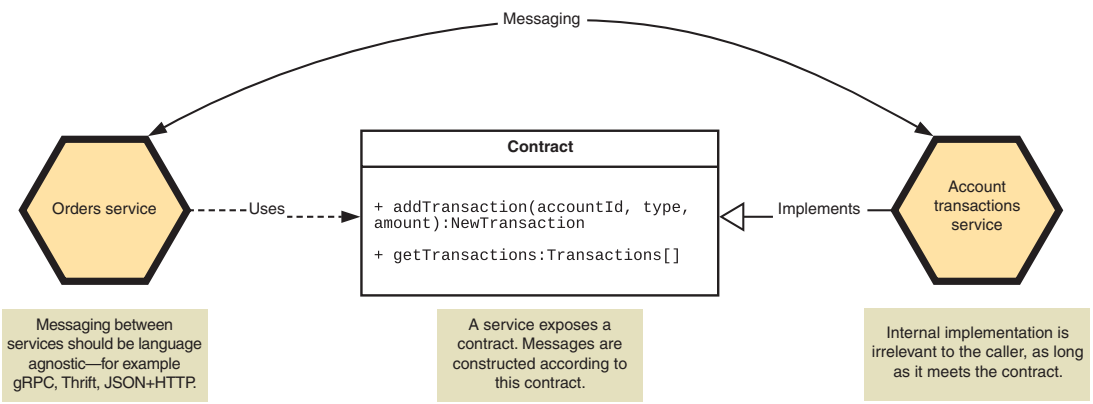


Figure 1.3 You can loosely couple services by having them communicate through defined contracts that hide implementation details.

Resilience

Microservices are a natural mechanism for isolating failure: if you deploy them independently, application or infrastructure failure may only affect part of your system. Likewise, being able to deploy smaller bits of functionality should help you change your system more gradually, rather than releasing a risky big bang of new functionality.

Consider the investment tool again. If the market service is unavailable, it won't be able to place the order to market. But a user can still request the order, and the service can pick it up later when the downstream functionality becomes available.

Although splitting your application into multiple services can isolate failure, it also will multiply points of failure. In addition, you'll need to account for what happens when failure *does* occur to prevent cascades. This involves both design—favoring asynchronous interaction where possible and using circuit breakers and timeouts appropriately—and operations—using provable continuous delivery techniques and robustly monitoring system activity.

Transparency

Most importantly, you need to know when a failure has occurred, and rather than one system, a microservice application depends on the interaction and behavior of multiple services, possibly built by different teams. At any point, your system should be transparent and observable to ensure that you both observe and diagnose problems.

Every service in your application will produce business, operational, and infrastructure metrics; application logs; and request traces. As a result, you'll need to make sense of a huge amount of data.

Automation

It might seem counterintuitive to alleviate the pain of a growing application by building a multitude of services. It's true that microservices are a more complex architecture than building a single application. By embracing automation and seeking consistency in the infrastructure *between* services, you can significantly reduce the cost of managing this additional complexity. You need to use automation to ensure the correctness of deployments and system operation.

It's not a coincidence that the popularity of microservice architecture parallels both the increasing mainstream adoption of DevOps techniques, especially *infrastructure-as-code*, and the rise of infrastructure environments that are fully programmable through APIs (such as AWS or Azure). These two trends have done a lot to make microservices feasible for smaller teams.

Alignment

Lastly, it's critical that you align your development efforts in the right way. You should aim to structure your services, and therefore your teams, around business concepts. This leads to higher cohesion.

To understand why this is important, consider the alternative. Many traditional SOAs deployed the technical tiers of an application separately—UI, business logic, integration, data. Figure 1.4 compares SOA and microservice architecture.

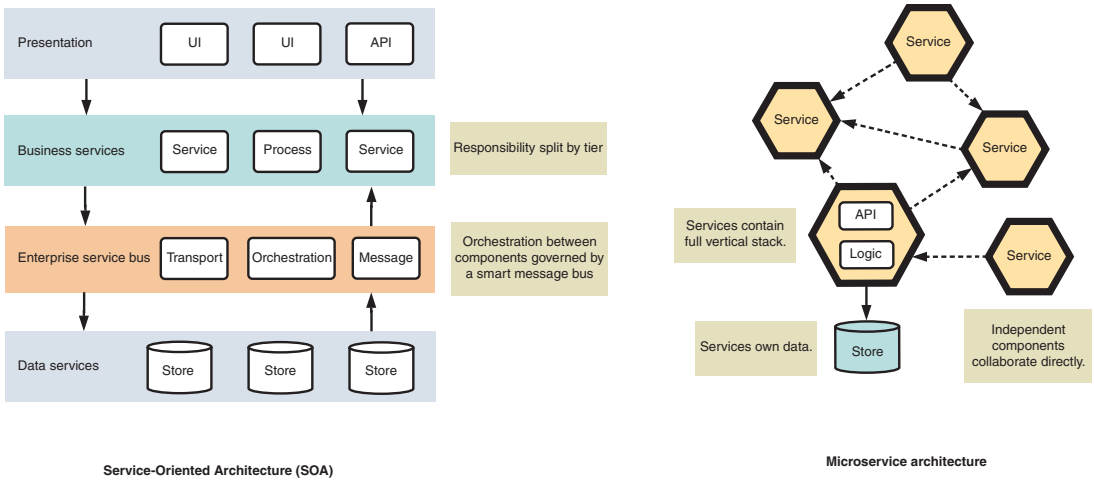


Figure 1.4 SOA versus microservice architecture

This use of *horizontal decomposition* in SOA is problematic, because cohesive functionality becomes spread across multiple systems. New features may require coordinated releases to multiple services and may become unacceptably coupled to others at the same level of technical abstraction.

A microservice architecture, on the other hand, should be biased toward vertical decomposition; each service should align to a single business capability, encapsulating all relevant technical layers.

> **NOTE** In rare instances, it might make sense to build a service that implements a technical capability, such as integration with a third-party service, if multiple services require it.

You also should be mindful of the consumers of your services. To ensure a stable system, you need to ensure you're developing patiently and maintaining backwards compatibility—whether explicitly or by running multiple versions of a service—to ensure that you don't force other teams to upgrade or break complex interactions between services.

Working with these five principles in mind will help you develop microservices well, leading to systems that are highly amenable to change, scalable, and stable.

1.1.3 Who uses microservices?

Many organizations have successfully built and deployed microservices, across many domains: in media (*The Guardian*); content distribution (SoundCloud, Netflix); transport and logistics (Hailo, Uber); e-commerce (Amazon, Gilt, Zalando); banking (Monzo); and social media (Twitter).

Most of these companies took a monolith-first approach.[1] They started by building a single large application, then progressively moved to microservices in response to growth pressures they faced. These pressures are outlined in Table 1.1.

Table 1.1 Pressures of growth on a software system

Pressure	Description
Volume	The volume of activity that a system performs may outgrow the capacity of original technology choices.
New features	New features may not be cohesive with existing features, or different technologies may be better at solving problems.
Engineering team growth	As a team grows larger, lines of communication increase. New developers spend more time comprehending the existing system and less time adding product value.
Technical debt	Increased complexity in a system—including debt from previous build decisions—increases the difficulty of making changes.
International distribution	International distribution may lead to data consistency, availability, and latency challenges.

For example, Hailo wanted to expand internationally—which would've been challenging with their original architecture—but also increase their pace of feature delivery.[2] SoundCloud wanted to be more productive, as the complexity of their original monolithic application was holding them back.[3] Sometimes, the shift coincided with a change in business priority: Netflix famously moved from physical DVD distribution to content streaming. Some of these companies completely decommissioned their original monolith. But for many, this is an ongoing process, with a monolith surrounded by a constellation of smaller services.

As microservice architecture has been more widely popularized—and as early adopters have open sourced, blogged, and presented the practices that worked for them—teams have increasingly begun greenfield projects using microservices, rather than building a single application first. For example, Monzo started with microservices as part of its mission to build a better and more scalable bank.[4]

[1] Martin Fowler expands on this pattern: "MonolithFirst," June 3, 2015, http://martinfowler.com/bliki/MonolithFirst.html.

[2] See Matt Heath, "A Long Journey into a Microservice World," *Medium*, May 30, 2015, http://mng.bz/XAOG.

[3] See Phil Calçado, "How we ended up with microservices," September 8, 2015, http://mng.bz/Qzhi.

[4] See Matt Heath, "Building microservice architectures in Go," June 18, 2015, http://mng.bz/9L83.

1.1.4 Why are microservices a good choice?

Plenty of successful businesses are built on monolithic software—Basecamp,[5] StackOverflow, and Etsy spring to mind. And in monolithic applications, a wealth of orthodox, long-established software development practice and knowledge exists. Why choose microservices?

TECHNICAL HETEROGENEITY LEADS TO MICROSERVICES

In some companies, technical heterogeneity makes microservices an obvious choice. At Onfido, we started building microservices when we introduced a product driven by machine learning—not a great fit for our original Ruby stack! Even if you're not fully committed to a microservice approach, applying microservice principles gives you a greater range of technical choices to solve business problems. Nevertheless, it's not always so clear-cut.

DEVELOPMENT FRICTION INCREASES AS COMPLEX SYSTEMS GROW

It comes down to the nature of complex systems. At the beginning of the chapter, we mentioned that software developers strive to craft effective and timely solutions to complex problems. But the software systems we build are inherently complex. No methodology or architecture can eliminate the essential complexity at the heart of such a system.

But that's no reason to get downhearted! You can ensure that the development approaches you take result in *good* complex systems, free from *accidental* complexity.

Take a moment and consider what you're trying to achieve as an enterprise software developer. Dan North puts it well:

> *The goal of software development is to sustainably minimize lead time to positive business impact.*

The hard part in complex software systems is to deliver sustainable value in the face of change: to continue to deliver with agility, pace, and safety even as the system becomes larger and more complex. Therefore, we believe a good complex system is one where two factors are minimized throughout the system's lifecycle: friction and risk.

Friction and risk limit your velocity and agility, and therefore your ability to deliver business impact. As a monolith grows, the following factors may lead to friction:

- Change cycles are coupled together, leading to higher coordination barriers and higher risk of regression.
- Soft module and context boundaries invite chaos in undisciplined teams, leading to tight or unanticipated coupling between components.
- Size alone can be painful: continuous integration jobs and releases—even local application startup—become slower.

These qualities aren't true for all monoliths, but unfortunately they're true for most that we've encountered. Likewise, these types of challenges are a common thread in the stories of the companies we mentioned.

---

[5] David Heinemeier Hansson coined the term "Majestic Monolith" to describe how 37signals built Basecamp: *Signal v. Noise*, February 29, 2016, http://mng.bz/1p3I.

Microservices reduce friction and risk

Microservices help reduce friction and risk in three ways:

- Isolating and minimizing dependencies at build time
- Allowing developers to reason about cohesive individual components, rather than an entire system
- Enabling the continuous delivery of small, independent changes

Isolating and minimizing dependencies at build time—whether between teams or on existing code—allows developers to move faster. Development can move in parallel, with reduced long-term dependency on past decisions made in a monolithic application. Technical debt is naturally limited to service boundaries.

Microservices are individually easier to build and reason about than monolithic applications. This is beneficial for the productivity of development in a growing organization. It also provides a compelling and flexible paradigm for coping with increased scale or smoothly introducing new technologies.

Small services are also a great enabler of continuous delivery. Deployments in large applications can be risky and involve lengthy regression and verification cycles. By deploying smaller elements of functionality, you better isolate changes to your active system, reducing the potential risk of an individual deployment.

At this point, we can come to two conclusions:

- Developing small, autonomous services can reduce friction in the development of long-running complex systems.
- By delivering cohesive and independent pieces of functionality, you can build a system that's malleable and resilient in the face of change, helping you to deliver sustainable business impact with reduced risk.

That doesn't mean everyone should build microservices. It'd be wonderful if there was an objective answer to the question "Do I need microservices?" but unfortunately you can only say "It depends"—on your team, on your company, and on the nature of the system you're building. If the scope of your system is trivial, then it's unlikely you'll gain benefits that outweigh the added complexity of building and running this type of fine-grained application. But if you've faced any of the challenges we mentioned earlier in this section, then microservices are a compelling solution.

A cautionary tale

We once heard a story about a microservice implementation gone wrong. The startup in question had begun to scale, and the CTO had decided that the only solution was to rebuild the application as microservices. If you're not worried by that sentence, you should be!

The engineering team set out to rebuild their application. This took them five months, during which time they released zero new features, nor did they release any of their microservices to production. The team proceeded to launch their new microservice application

(continued)

during the busiest month for the business, causing absolute chaos and necessitating a rollback to the original monolith.

This type of migration gives microservices a bad name. Few businesses have the luxury of a feature freeze for several months nor can they indulge a big-bang launch of a new architecture. Although the sample set is small, most successful microservice migrations that we've observed have been piecemeal, balancing architectural vision with business needs, priorities, and resource constraints. Although it'll take longer and require more engineering effort, hopefully you'll never recognize your team being mentioned in a cautionary tale!

1.2 What makes microservices challenging?

Let's dig a little deeper and explore the costs and complexity of designing and running microservices. Microservices aren't the only architecture that have promised nirvana through decomposition and distribution, but those past attempts, such as SOA,[6] are widely considered unsuccessful. No technique is a silver bullet. For example, as we've mentioned, microservices drastically increase the number of moving parts in a system. By distributing functionality and data ownership across multiple autonomous services, you likewise distribute responsibility for stability and sane operation of your application.

You'll encounter many challenges when designing and running a microservice application:

- *Scoping and identifying microservices* requires substantial domain knowledge.
- The right *boundaries and contracts* between services are difficult to identify and, once you've established them, can be time-consuming to change.
- Microservices are *distributed systems* and therefore require different assumptions to be made about state, consistency, and network reliability.
- By distributing system components across networks, and increasing technical heterogeneity, microservices introduce *new modes of failure*.
- It's more challenging to understand and verify what should happen in normal operation.

1.2.1 Design challenges

How do these challenges impact the design and runtime phases of microservice development? Earlier we introduced the five key principles underlying microservice development. The first of those was *autonomy*. For your services to be autonomous, you need to design them such that, together, they're loosely coupled, and, individually, they encapsulate highly cohesive elements of functionality. This is an evolutionary process. The

[6] SOA is a wooly term. Although many principles of SOA are similar to microservices, the definition of the former is inextricably associated with heavyweight, enterprise vendor tools, such as ESBs.

scope of your services may change over time, and you'll often choose to carve out new functionality from—or even retire—existing services.

Making those choices is challenging, and even more so at the start of developing an application! The primary driver of loose coupling is the boundaries you establish between services; getting those wrong will lead to services that are resistant to change and, overall, a less malleable and flexible application.

Scoping microservices requires domain knowledge

Each microservice is responsible for a single capability. Identifying these capabilities requires knowledge of the business domain of your application. Early in an application's lifetime, your domain knowledge might be at best incomplete, or at worst, incorrect.

Inadequate understanding of your problem domain can result in poor design choices. In a microservice application, the increased rigidity of a service boundary when compared to a module within a monolithic application means the downstream cost of poor scoping decisions is likely to be higher:

- You may need to refactor across multiple distinct codebases.
- You may need to migrate data from one service's database to another.
- You may not have identified implicit dependencies between services, which could lead to errors or incompatibility on deployment.

These activities are illustrated in figure 1.5.

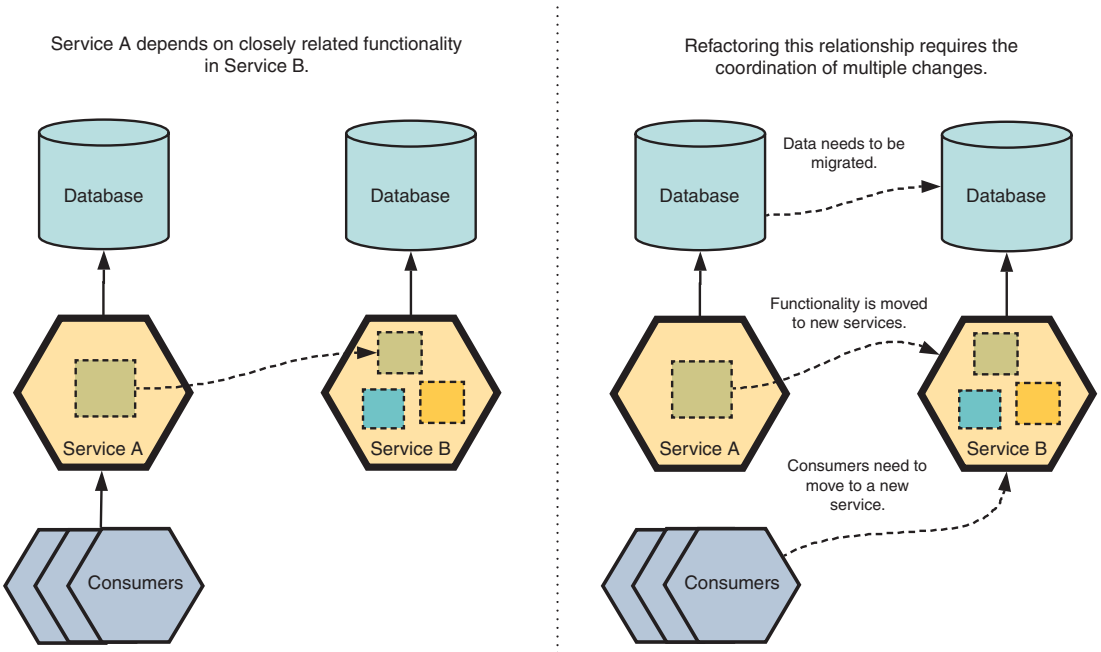


Figure 1.5 Incorrect service scoping decisions may require complex and costly refactoring across service boundaries.

But making design decisions based on insufficient domain knowledge is hardly unique to microservices! The difference is in the impact of those decisions.

NOTE In chapters 2 and 4, we'll discuss best practices for identifying and scoping services, using an example application.

MAINTAINING CONTRACTS BETWEEN SERVICES

Each microservice should be independent of the implementation of other services. This enables technical heterogeneity and autonomy. For this to work, each microservice should expose a *contract*—analogous to an interface in object-oriented design—defining the messages it expects to receive and respond with. A good contract should be

- *Complete*—Defines the full scope of an interaction
- *Succinct*—Takes in no more information than is necessary, so that consumers can construct messages within reasonable bounds
- *Predictable*—Accurately reflects the real behavior of any implementation

Anyone who's designed an API might know how hard these properties are to achieve. Contracts become the glue between services. Over time, contracts may need to evolve while also needing to maintain backwards compatibility for existing collaborators. These twin tensions—between stability and change—are challenging to navigate.

MICROSERVICE APPLICATIONS ARE DESIGNED BY TEAMS

In larger organizations, it's likely that multiple teams will build and run a microservice application, each taking responsibility for different microservices. Each team may have its own goals, way of working, and delivery lifecycle. It can be difficult to design a cohesive system when you also need to reconcile the timelines and priorities of other independent teams. Coordinating the development of any substantial microservice application therefore will require the agreement and reconciliation of priorities and practices across multiple teams.

MICROSERVICE APPLICATIONS ARE DISTRIBUTED SYSTEMS

Designing microservice applications means designing distributed systems. Many fallacies occur in the design of distributed systems,[7] including

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- Transport cost is zero.

Clearly, assumptions you might make in nondistributed systems—such as the speed and reliability of method calls—are no longer appropriate and can lead to poor, unstable implementation. You must consider latency, reliability, and the consistency of state across your application.

[7] See Arnon Rotem-Gal-Oz, "Fallacies of Distributed Computing Explained," https://pages.cs.wisc.edu/~zuyu/files/fallacies.pdf.

Once the application is distributed—where the application's underlying state data is spread across a multitude of places—consistency becomes challenging. You may not have guarantees of the order of operations. It won't be possible to maintain ACID-like transactional guarantees when actions take place across multiple services. This will affect design at the application level: you'll need to consider how a service might operate in an inconsistent state and how to roll back in the event of transaction failure.

1.2.2 Operational challenges

A microservice approach will inherently multiply the possible points of failure in a system. To illustrate this, let's return to the investment tool we mentioned earlier. Figure 1.6 identifies possible points of failure in this application. You can see that something could go wrong in multiple places, and that could affect the normal processing of an order.

Consider the questions you might need to answer when this application is in production:

- If something goes wrong and your user's order isn't placed, how would you determine where the fault occurred?
- How do you deploy a new version of a service without affecting order placement?
- How do you know which services were meant to be called?
- How do you test that this behavior is working correctly across multiple services?
- What happens if a service is unavailable?

Rather than eliminating risk, microservices move that cost to later in the lifecycle of your system: reducing friction in development but increasing the complexity of how you deploy, verify, and observe your application in operation.

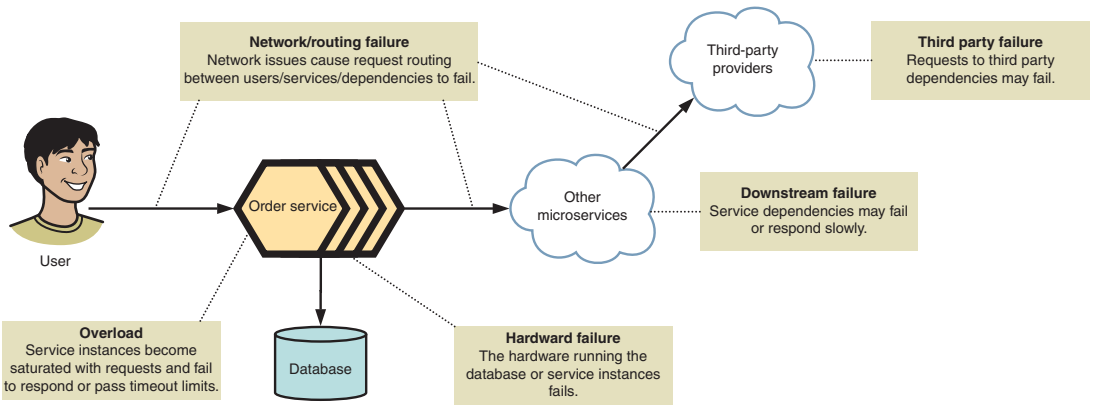


Figure 1.6 Possible points of failure when placing a sell order

A microservices approach suggests an evolutionary approach to system design: you can add new features independently without changing existing services. This minimizes the cost and risk of change.

But in a decoupled system that constantly changes, it can be extremely difficult to keep track of the big picture, which makes issue diagnosis and support more challenging. When something goes wrong, you need to have some way of tracing how the system *did* behave (what services it called, in which order, and what the outcome was), but you also need some way of knowing how the system *should have* behaved.

Ultimately, you face two operational challenges in microservices: observability and multiple points of failure. Let's focus on each of those in turn.

OBSERVABILITY IS DIFFICULT TO ACHIEVE

We touched on the importance of transparency back in section 1.1.2. But why is it harder in microservice applications? It's harder because you need to understand the big picture. You need to assemble that big picture from multiple jigsaw pieces, to correlate and link together the data each service produces to ensure you understand what each service does within the wider context of delivering some business output. Individual service logs provide a partial view of system operation, which is helpful, but you need to use both a microscope and a wide-angle lens to understand the system in full.

Likewise, because you're running multiple applications, depending on how you choose to deploy them, a less obvious correlation may exist between underlying infrastructural metrics—like memory and CPU usage—and the application. These metrics are still useful but are less of a focus than they might be in a monolithic system.

MULTIPLYING SERVICES MULTIPLIES POINTS OF FAILURE

We're probably not being too pessimistic if we say that everything that can fail will fail. It's important that you start with that mindset: if you assume weakness and fragility in the multiple services forming your system, that can better inform how you design, deploy, and monitor that system—rather than getting too surprised when something does go wrong.

You need to consider how your system will continue operating despite the failures of individual components. This implies that, individually, services will need to become more robust—considering error checking, failover, and recovery—but also that the whole system should act reliably, even when individual components are never 100% reliable.

1.3 Microservice development lifecycle

At an individual level, each microservice should look familiar to you—even if it's a bit smaller. To build a microservice, you'll use many of the same frameworks and techniques that you'd normally apply in building an application: web application frameworks, SQL databases, unit tests, libraries, and so on.

At a system level, choosing a microservice architecture will have a significant impact on how you design and run your application. Throughout this book, we'll focus on these

three key stages in the development lifecycle of a microservice application: designing services, deploying them to production, and observing their behavior. This cycle is illustrated in figure 1.7.

Making well-reasoned decisions in each of these three stages will help you build applications that are resilient, even in the face of changing requirements and increasing complexity. Let's walk through each stage and consider the steps you'll take to deliver an application with microservices.

1.3.1 Designing microservices

You'll need to make several design decisions when building a microservice application that you wouldn't have encountered building monolithic apps. The latter often follow well-known patterns or frameworks, such as three-tier architecture or model-view controller (MVC). But techniques for designing microservices are still in their relative infancy. You'll need to consider

- Whether to start with a monolith or commit to microservices up front
- The overall architecture of your application and the façade it presents to outside consumers
- How to identify and scope the boundaries of your services
- How your services communicate with each other, whether synchronously or asynchronously
- How to achieve resiliency in services

That's quite a lot of ground to cover. For now, we'll touch on each of these considerations so you can see why paying attention to all of them is vital to a well-designed microservice application.

MONOLITH FIRST?

You'll find two opposing trends to starting with microservices: monolith first or microservices only. Advocates of the former reason that you should always start with a monolith, as you won't understand the component boundaries in your system at an early stage, and the cost of getting these wrong is much higher in a microservice application. On the other hand, the boundaries you choose in a monolith aren't necessarily the same ones you'd choose in a well-designed microservice application.

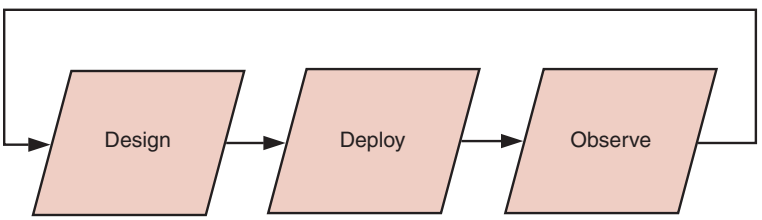


Figure 1.7 The key iterative stages—design, deploy, and observe—in the microservice development lifecycle

Although the speed of development may be slower to begin with, microservices will reduce friction and risk in future development. Likewise, as tooling and frameworks mature, microservices best practice is becoming increasingly less daunting to pick up. Either way you want to go, the advice in this book should be useful, regardless of whether you're thinking of migrating away from your monolith or starting afresh.

SCOPING SERVICES

Choosing the right level of responsibility for each service—its scope—is one of the most difficult challenges in designing a microservice application. You'll need to model services based on the business capabilities they provide to an organization.

Let's extend the example from the beginning of this chapter. How might your services change if you wanted to introduce a new, special type of order? You have three options to solve this problem (figure 1.8):

1. Extend the existing service interface
2. Add a new service endpoint
3. Add a new service

Each of these options has pros and cons that will impact the cohesiveness and coupling between services in your application.

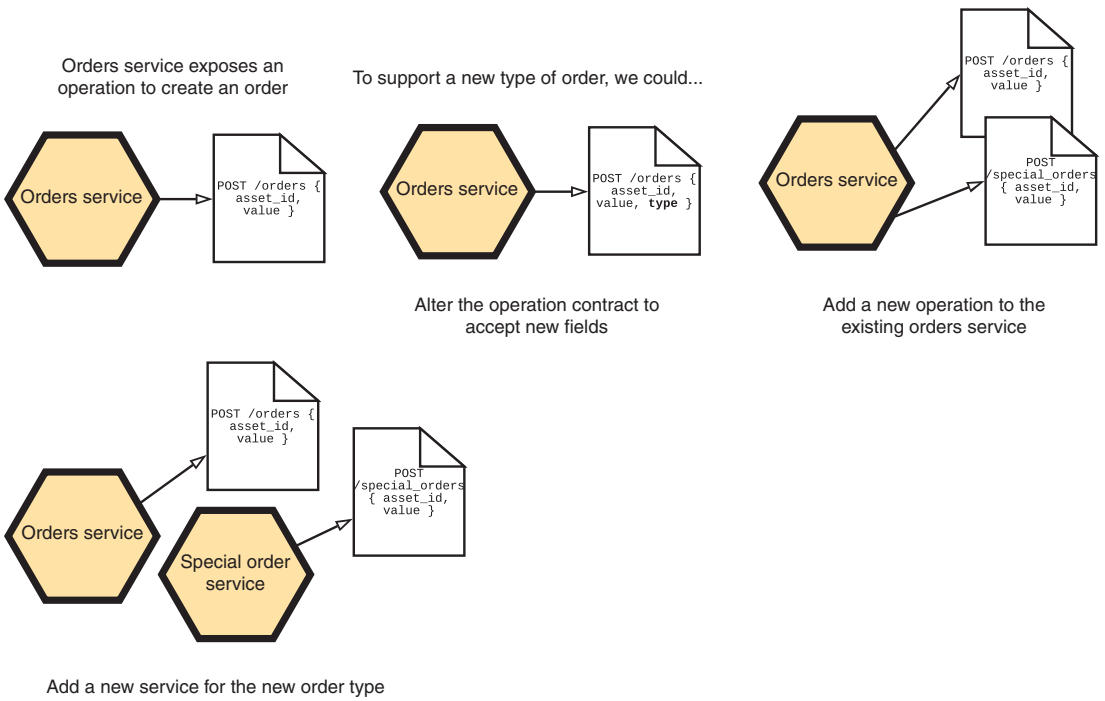


Figure 1.8. To scope functionality, you need to make decisions about whether capabilities belong in existing services or if you need to design new services.

NOTE In chapters 2 and 4, we'll explore service scoping and how to make optimal decisions about service responsibility.

COMMUNICATION

Communication between services may be asynchronous or synchronous. Although synchronous systems are easier to reason through, asynchronous systems are highly decoupled—reducing the risk of change—and potentially more resilient. But the complexity of such a system is high. In a microservice application, you need to balance synchronous and asynchronous messaging to choreograph and coordinate the actions of multiple microservices effectively.

RESILIENCY

In a distributed system, a service can't trust its collaborators, not necessarily because they're coded poorly or because of human error, but because you can't safely assume the network between or behavior of those services is reliable or predictable. Services need to be resilient in the face of failure. To achieve this, you need to design your services to work defensively by backing off in the event of errors, limiting request rates from poor collaborators, and dynamically finding healthy services.

1.3.2 Deploying microservices

Development and operations must be closely intertwined when building microservices. It's not going to work if you build something and throw it over the fence for someone else to deploy and operate it. In a system composed of numerous, autonomous services, if you build it, you should run it. Understanding how your services run will in turn help you make better design decisions as your system grows.

Remember, what's special about your application is the business impact it delivers. That emerges from collaboration between multiple services. In fact, you could standardize or abstract away anything outside of the unique capability each service offers—ensuring teams are focused on business value. Ultimately, you should reach a stage where there's no ceremony involved in deploying a new service. Without this, you'll invest all your energy in plumbing, rather than creating value for customers.

In this book, we'll teach you how to construct a reliable road to production for existing and new services. The cost of deploying new services must be negligible to enable rapid innovation. Likewise, you should standardize this process to simplify system operation and ensure consistency across services. To achieve this, you'll need to

- Standardize microservice deployment artifacts
- Implement continuous delivery pipelines

We've heard reliable deployment described as boring, not in the sense that it's unexciting, but that it's incident-free. Unfortunately, we've seen too many teams where the opposite is true: deploying software is stressful and encourages unhealthy all-hands-on-deck behavior. This is bad enough for one service—if you're deploying any number

of services, the anxiety alone will drive you mad! Let's look at how these steps lead to stable and reliable microservice deployments.

STANDARDIZE MICROSERVICE DEPLOYMENT ARTIFACTS

It often seems like every language and framework has its own deployment tool. Python has Fabric, Ruby has Capistrano, Elixir has exrm, and so on. And then the deployment environment itself is complex:

- What server does an application run on?
- What are the application's dependencies on other tools?
- How do you start that application?

At runtime, an application's dependencies (figure 1.9) are broad and might include libraries, binaries and OS packages (such as ImageMagick or libc), and OS processes (such as cron or fluentd).

Technically, heterogeneity is a fantastic benefit of service autonomy. But it doesn't make life easy for deployment. Without consistency, you won't be able to standardize your approach to taking services to production, which increases the cost of managing deployments and introducing new technology. At worst, each team reinvents the wheel, coming up with different approaches for managing dependencies, packing builds, getting them onto servers, and operating the application itself.

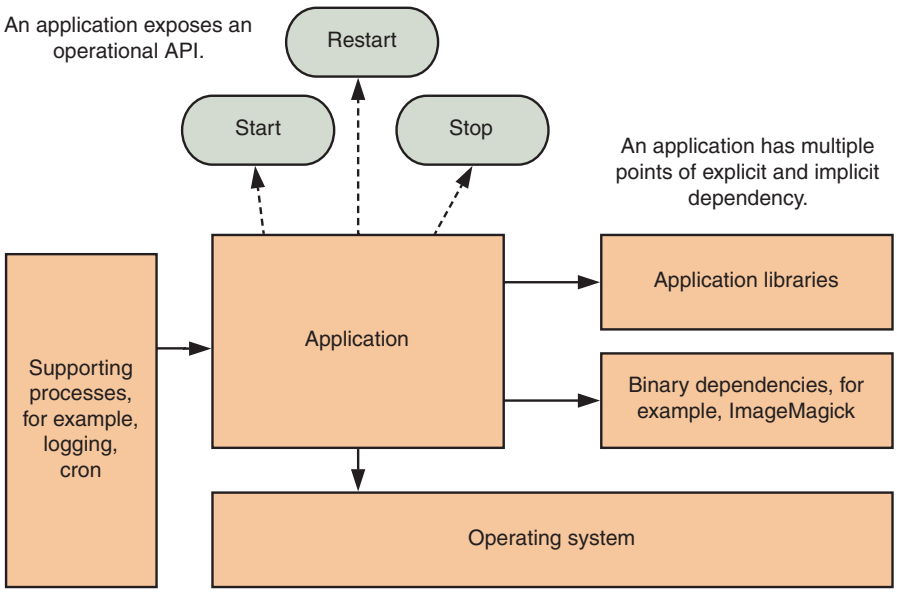


Figure 1.9 An application exposes an operational API and has many types of dependencies, including libraries, binary dependencies, and supporting processes.

Our experience suggests the best tools for this job are *containers*. A container is an operating system-level virtualization method that supports running isolated systems on a host, each with its own network and process space, sharing the same kernel. A container is quicker to build and quicker to start up than a virtual machine (seconds, rather than minutes). You can run multiple containers on one machine, which simplifies local development and can help to optimize resource usage in cloud environments.

Containers standardize the packaging of an application, and the runtime interface to it, and provide immutability of both operating environment and code. This makes them powerful building blocks for higher level composition. By using them, you can define and isolate the full execution environment of any service.

Although many implementations of containers are available (and the concept exists outside of Linux, such as jails in FreeBSD and zones in Solaris), the most mature and approachable tooling that we've used so far is Docker. We'll use that tool later in this book.

IMPLEMENT CONTINUOUS DELIVERY PIPELINES

Continuous delivery is a practice in which developers produce software that they can reliably release to production at any time. Imagine a factory production line: to continuously deliver software, you build similar pipelines to take your code from commit to live operation. Figure 1.10 illustrates a simple pipeline. Each stage of the pipeline provides feedback to the development team on the correctness of their code.

Earlier, we mentioned that microservices are an ideal enabler of continuous delivery because their smaller size means you can develop them quickly and release them independently. But continuous delivery doesn't automatically follow from developing microservices. To continuously deliver software, you need to focus on two goals:

- Building a set of validations that your software has to pass through. At each stage of your deployment process, you should be able to prove the correctness of your code.
- Automating the pipeline that delivers your code from commit to production.

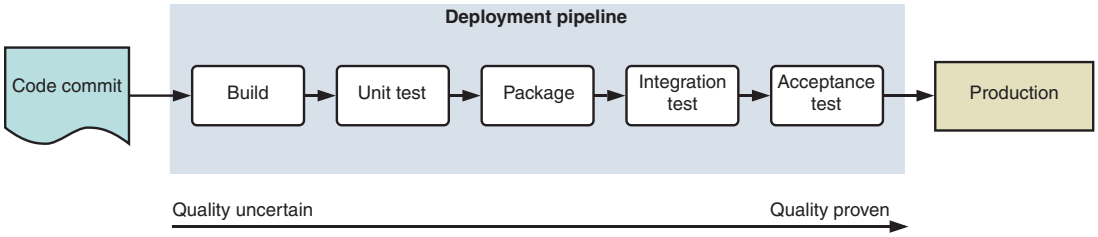


Figure 1.10 A high-level deployment pipeline for a microservice

Building a provably correct deployment pipeline will allow developers to work safely and at pace as they iteratively develop services. Such a pipeline is a repeatable and reliable process for delivering new features. Ideally, you should be able to standardize the validations and steps in your pipeline and use them across multiple services, further reducing the cost of deploying new services.

Continuous delivery also reduces risk, because the quality of the software produced and the team's agility in delivering changes are both increased. From a product perspective, this may mean you can work in a leaner fashion—rapidly validating your assumptions and iterating on them.

> **NOTE** In part 3, we'll build a continuous delivery pipeline using the Pipeline feature of the freely available Jenkins continuous integration tool. We'll also explore different deployment patterns, such as canaries and blue-green deployments.

1.3.3 Observing microservices

We've discussed transparency and observability throughout this chapter. In production, you need to know what's going on. The importance of this is twofold:

- You want to proactively identify and refactor fragile implementation in your system.
- You need to understand how your system is behaving.

Thorough monitoring is significantly more difficult in a microservice application because single transactions may span multiple distinct services; technically heterogeneous services might produce data in irreconcilable formats; and the total volume of operational data is likely to be much higher than that of a single monolithic application. But if you're able to understand how your system operates—and observe that closely—despite this complexity, you'll be better placed to make effective changes to your system.

IDENTIFY AND REFACTOR POTENTIALLY FRAGILE IMPLEMENTATION

Systems will fail, whether because of bugs introduced, runtime errors, network failures, or hardware problems.[8] Over time, the cost of eliminating unknown bugs and errors becomes higher than the cost of being able to react quickly and effectively when they occur.

Monitoring and alerting systems allow you to diagnose problems and determine what causes failures. You may have automated mechanisms reacting to the alerts that'll spawn new container instances in different data centers or react to load issues by increasing the number of running instances of a service.

To minimize the consequences of those failures, and prevent them cascading throughout the system, you need to be able to architect dependencies between services in ways

[8] You even have to watch out for squirrels: Rich Miller, "Surviving Electric Squirrels and UPS Failures," *DataCenter Knowledge*, July 9, 2012, http://mng.bz/rmbF.

that'll allow for partial degradation. One service going down shouldn't bring down the whole application. It's important to think about the possible failure points of your applications, recognize that failure will always happen, and prepare accordingly.

UNDERSTAND BEHAVIOR ACROSS HUNDREDS OF SERVICES

You need to prioritize transparency in design and implementation to understand behavior across your services. Collecting logs and metrics—and unifying them for analytical and alerting purposes—allows you to build a single source of truth to resort to when monitoring and investigating the behavior of your system.

As we mentioned in section 1.3.2, you can standardize and abstract anything outside of the unique capability each service offers. You can think of each service as an onion. At the center of that onion, you have the unique business capability offered by that service. Surrounding that, you have layers of instrumentation—business metrics, application logs, operational metrics, and infrastructure metrics—that make that capability observable. You can then trace each request to the system through these layers. You'd then push the data you collected from these layers to an operational data store for analytics and alerting. This is illustrated in figure 1.11.

> **NOTE** In part 4 of this book, we'll discuss how to build a monitoring system for microservices, collect appropriate data, and use that data to produce a live model for a complex microservice application.

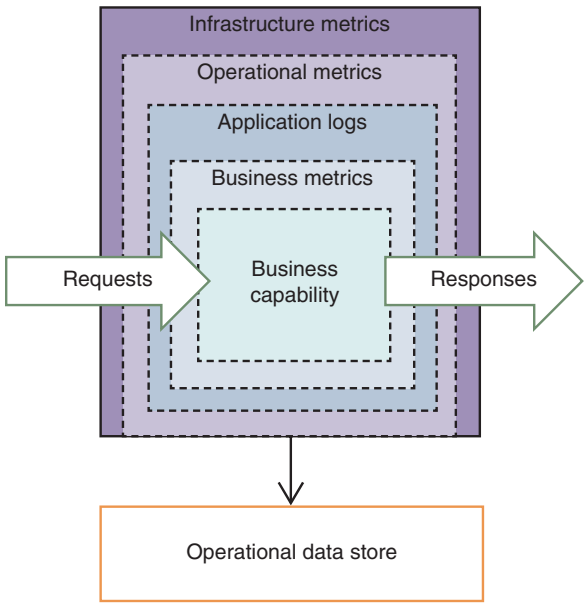


Figure 1.11 A business capability microservice surrounded by layers of instrumentation, through which pass requests to the microservice and its responses, with data collected from the process going to an operational data store

1.4 Responsible and operationally aware engineering culture

It'd be a mistake to examine the technical nature of microservices in isolation from how an engineering team works to develop them. Building an application out of small, independent services will drastically change how an organization approaches engineering, so guiding the culture and priorities of your team will be a significant factor in whether you successfully deliver a microservice application.

It can be difficult to separate cause and effect in organizations that have successfully built microservices. Was the development of fine-grained services a logical outcome of their organizational structure and the behavior of their teams? Or did that structure and behavior arise from their experiences building fine-grained services?

The answer is a bit of both. A long-running system isn't only an accumulation of features requested, designed, and built. It also reflects the preferences, opinions, and objectives of its builders and operators. Conway's Law expresses this to some degree:

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

"Constrained" might suggest that these communication structures will limit and constrict the effective development of a system. In fact, microservices practice implies the opposite: that a powerful way to avoid friction and tension in building systems is to design an organization in the shape of the system you intend to build.

Deliberate symbiosis with organizational structure is one example of common microservices practice. To be able to realize benefits from microservices and adequately manage their complexity, you need to develop working principles and practices that are effective for that type of application, rather than using the same techniques that you used to build monoliths.

Summary

- Microservices are both an architectural style and a set of cultural practices, underpinned by five key principles: autonomy, resilience, transparency, automation, and alignment.
- Microservices reduce friction in development, enabling autonomy, technical flexibility, and loose coupling.
- Designing microservices can be challenging because of the need for adequate domain knowledge and balancing priorities across teams.
- Services expose contracts to other services. Good contracts are succinct, complete, and predictable.
- Complexity in long-running software systems is unavoidable, but you can deliver value sustainably in these systems if you make choices that minimize friction and risk.
- Reliably incident-free ("boring") deployment reduces the risk of microservices by making releases automated and provable.

- Containers abstract away differences between services at runtime, simplifying large-scale management of heterogeneous microservices.
- Failure is inevitable: microservices need to be transparent and observable for teams to proactively manage, understand, and own service operation ... and the lack thereof.
- Teams adopting microservices need to be operationally mature and focus on the entire lifecycle of a service, not only on the design and build stages.

SOFTWARE DEVELOPMENT

Microservices IN ACTION

Bruce • Pereira

Invest your time in designing great applications, improving infrastructure, and making the most out of your dev teams. Microservices are easier to write, scale, and maintain than traditional enterprise applications because they're built as a system of independent components. Master a few important new patterns and processes, and you'll be ready to develop, deploy, and run production-quality microservices.

Microservices in Action teaches you how to write and maintain microservice-based applications. Created with day-to-day development in mind, this informative guide immerses you in real-world use cases from design to deployment. You'll discover how microservices enable an efficient continuous delivery pipeline, and explore examples using Kubernetes, Docker, and Google Container Engine.

What's Inside

- An overview of microservice architecture
- Building a delivery pipeline
- Best practices for designing multi-service transactions and queries
- Deploying with containers
- Monitoring your microservices

Written for intermediate developers familiar with enterprise architecture and cloud platforms like AWS and GCP.

Morgan Bruce and **Paulo A. Pereira** are experienced engineering leaders. They work daily with microservices in a production environment, using the techniques detailed in this book.

"The one [and only] book on implementing microservices with a real-world, cover-to-cover example you can relate to."
—Christian Bach, Swiss Re

"A perfect fit for those who want to move their majestic monolith to a scalable microservice architecture."
—Akshat Paul
McKinsey & Company

"Shows not only how to write microservices, but also how to prepare your business and infrastructure for this change."
—Maciej Jurkowski, Grupa Pracuj

"A deep dive into microservice development with many real and useful examples."
—Antonio Pessolano
Consoft Sistemi

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/microservices-in-action

ISBN-13: 978-1-61729-445-7
ISBN-10: 1-61729-445-4
54999
9 781617 294457

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]