Microservices IN ACTION

Morgan Bruce
Paulo A. Pereira

MANNING

MEAP Edition
Manning Early Access Program
Microservices in Action
Version 3

welcome

Thank you for purchasing the MEAP for *Microservices in Action*. We're very excited to release these first few chapters and we are looking forward to completing the book. This book is a pragmatic guide for engineers who want to build and run well-designed, stable and resilient microservice applications.

Microservices – applications built from loosely-coupled, autonomous services – can help engineering teams deliver complex, long-running applications at pace without sacrificing maintainability. Developing microservices drastically changed how we approached designing, deploying and monitoring our applications; with this book we're hoping to share the lessons we learnt.

You'll find the first three chapters of the book in this first MEAP release. In these chapters, we explore approaches for designing services and architecting microservice applications, as well as our thoughts on the underlying architectural principles and common challenges of microservice development.

This book consists of four parts. Looking ahead, later chapters will further explore the design, deployment and monitoring of microservice applications. The rest of Part 2 will examine functional decomposition, microservice messaging, designing for reliability and service frameworks.

Part 3 of this book covers deployment. We'll demonstrate how you can get deployment right for your services: being able to deploy rapidly and reliably is the difference between success and failure for many microservice applications.

And in Part 4, we'll examine how microservices increase the complexity of monitoring, and how to build the tools you'll need to support debugging, alerting and understanding system behaviour.

Lastly, we hope you'll take advantage of the Author Online Forum. We'll be reading your comments and responding throughout the MEAP. We deeply appreciate your feedback – it will help us write a better book!

—Morgan Bruce and Paulo A Pereira

brief contents

1 Designing and Running Microservices

This chapter covers:

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

Software developers strive to craft effective and timely solutions to complex problems. The first problem we usually try and solve is: what does our customer want? If we're skilled (or lucky), we get that right. But our efforts rarely stop there. Our successful application continues to grow: we debug issues; we build new features; we 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, our once simple and stable product becomes both intractable and delicate: instead of sustainably delivering more value to our customers, we are fatigued from outages, anxious about releasing, and too slow to deliver new features or fixes. Neither our customers nor our developers are happy.

Microservices promise a better way to sustainably deliver business impact. Rather than a single monolithic unit, applications built in this style are composed from loosely-coupled, autonomous services. By building services that "do one thing well", we can avoid the inertia and entropy of large applications. Even in existing applications, we can progressively extract functionality into independent services to make our 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 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 that, individually, "do one thing well", but 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 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, we try and design for these properties at a class, module or library level. In a microservice application, we 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 microservices 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 e.g. shares
- Modelling 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 microservices application.

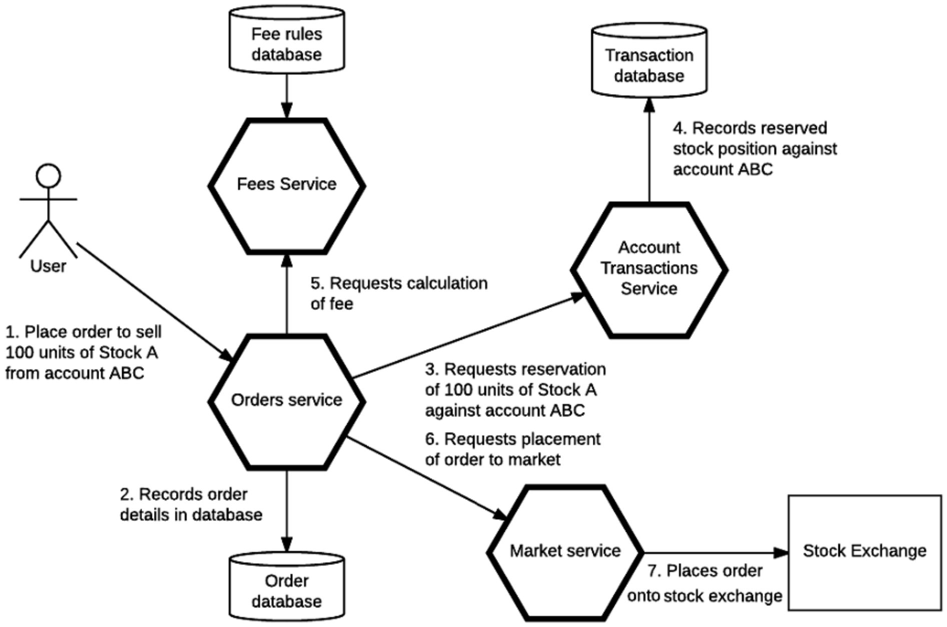
Fee rules database
Transaction database
4. Records reserved stock position against account ABC
Fees Service
User
Account Transactions Service
5. Requests calculation of fee
1. Place order to sell 100 units of Stock A from account ABC
Orders service
3. Requests reservation of 100 units of Stock A against account ABC
6. Requests placement of order to market
2. Records order details in database
Market service
Stock Exchange
7. Places order onto stock exchange
Order database

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

We can observe three key characteristics of microservices in this figure:

- Each microservice is *responsible for a single capability.* This might be a 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, as other services can only access data they don't own through the interface that a service provides.
- Choreography and collaboration – the sequencing of messages and actions to perform some useful activity - are the responsibility of services themselves, not the messaging mechanism that connects them, nor another piece of software.

In addition to these three characteristics, we 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 this 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 above system helps us 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, we need to alert our risk team if it doesn't match normal trading patterns. It would 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

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

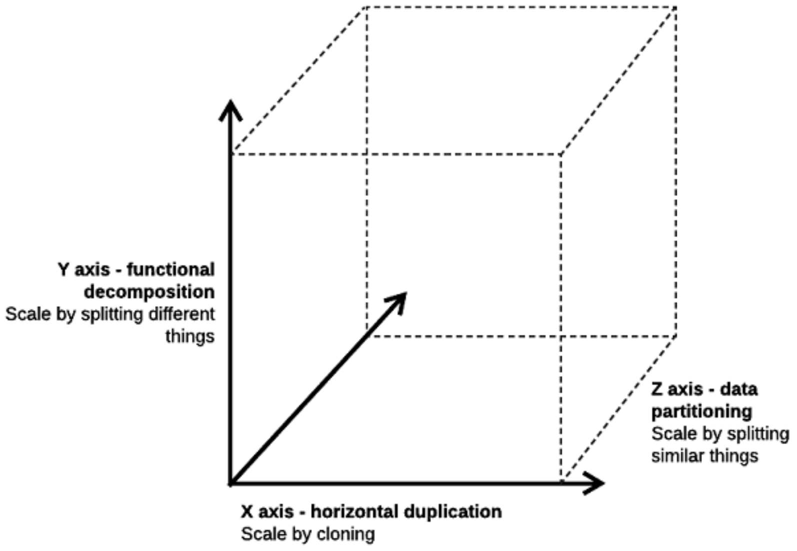
Y axis - functional decomposition
Scale by splitting different things
Z axis - data partitioning
Scale by splitting similar things
X axis - horizontal duplication
Scale by cloning

Figure 1.2 The three dimensions of scaling an application

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 a system may be decomposed to address the unique scale needs of different functionality.

Let's revisit our investment tool as an example:

- Financial predictions might be computationally onerous, but rarely done
- Investment accounts may be governed by complex regulatory and business rules
- Market trading may happen in extremely large volumes, but also rely on minimizing latency

If we build these features as microservices, we 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 means that their underlying resource needs can be managed separately. Interestingly, this also implies a natural way to limit failure: if our financial prediction service fails, that failure is unlikely to cascade to the market trading or investment account services.

So, microservice applications have some interesting technical properties:

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

Beyond these properties, we believe there are underlying principles that should drive your technical and organizational decisions when building and running a microservice application. Let's explore them now.

1.1.2 Key principles

There are five cultural and architectural principles that underpin microservices development:

- Autonomy
- Resilience
- Transparency
- Automation
- Alignment

Let's look at each of these principles in turn.

AUTONOMY

We've established that microservices are *autonomous*. That is, each service operates and changes independently of others. To ensure that autonomy, we need to design our services so that they're:

- *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 Order service we introduced earlier should not be aware of the implementation of the Account Transaction service. This is illustrated in

Figure 1.3.

- *Independently deployable*: services will be developed in parallel, often by multiple teams. Being forced to deploy these in lock-step or in an orchestrated formation would result in risky and anxious deployments. Ideally we want to use our smaller services to enable rapid, frequent and small releases.

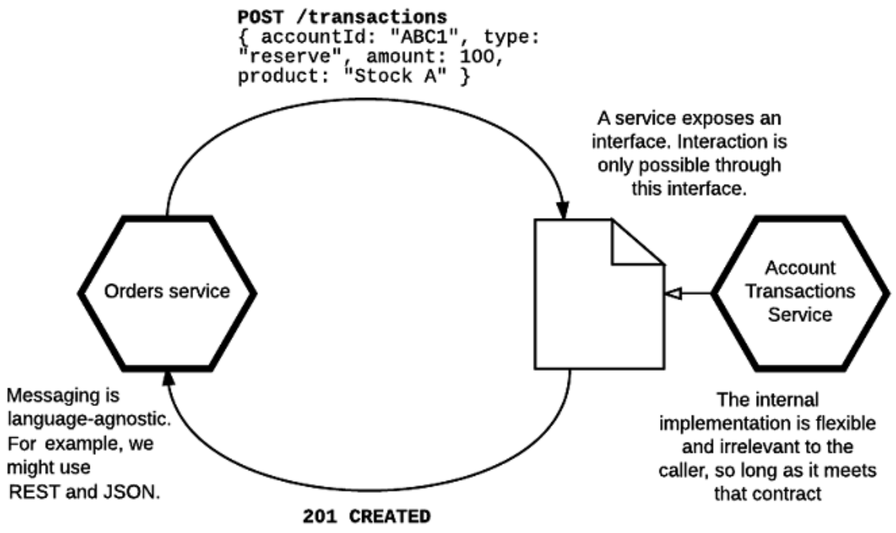
POST /transactions
{ accountId: "ABC1", type: "reserve", amount: 100, product: "Stock A" }
A service exposes an interface. Interaction is only possible through this interface.
Orders service
Account Transactions Service
Messaging is language-agnostic. For example, we might use REST and JSON.
201 CREATED
The internal implementation is flexible and irrelevant to the caller, so long as it meets that contract

Figure 1.3 Services may be loosely coupled by communicating through defined contracts that hide implementation details

Autonomy is also cultural. It's vital that accountability for and ownership of services is delegated to teams responsible for delivering business impact. As we've established, organizational design has an influence on system design. Owning their own services 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 4 we'll discuss developing responsible and autonomous engineering teams, and why this is crucial when working with microservices.

RESILIENCE

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

Consider our investment tool again. If the Market service is unavailable, the order won't be placed to market. However, this still means that the order can still be requested by a user, and picked up later when the downstream functionality becomes available.[1]

Conversely, splitting out our application into multiple services will multiply points of failure. This also means that we need to account for what happens when failure *does* occur in order to prevent cascades. This involves both design – favoring asynchronous interaction where possible, using circuit breakers and timeouts appropriately – and operations – using provable continuous delivery techniques and robustly monitoring system activity.

TRANSPARENCY

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

Every service in our application will produce business, operational and infrastructure metrics, application logs and request traces – so we'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 we can significantly reduce the cost of managing this additional complexity.

Crucially, we need to use automation to ensure the correctness of deployments and the correctness of system operation. We'll explore this in depth in Part 3.

ALIGNMENT

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

To understand why this is important, let's examine the alternative. Many traditional SOAs deployed the technical tiers of an application separately – UI, business logic, integration, data. We can call this horizontal decomposition. This is problematic, as cohesive functionality becomes spread across multiple systems. New features may require coordinated releases to

[1] **Although if the user was aiming to get a particular price on their stock, this might not actually be a good idea!**

multiple services; new features may become unacceptably coupled to others at the same level of technical abstraction.

Instead, a microservices system 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 this is required by multiple services.

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

Working with these five principles in mind will help us develop microservices well, leading to systems that are highly amenable to change, stable and scalable. Now that we've established some key principles behind microservices, let's explore why they're a compelling technique for the development of long-running complex applications.

1.1.3 Why use microservices?

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

Most of these companies took a "monolith-first" approach[2], progressively moving to microservices in response to growth pressures they faced. These pressures are outlined in Table 1.2.

Table 1.2 Pressures of growth on a software system

Pressure	Description
Volumes	The volume of activity performed by a system may outgrow the capacity of original technology choices
International expansion	International expansion is similar to growth in processing volumes, but may also lead to data consistency, availability and latency challenges
New features	New features may not be cohesive with existing features, or may be better solved by different technologies

[2] Martin Fowler expands on this pattern: http://martinfowler.com/bliki/MonolithFirst.html

Engineering team growth	As a team grows larger, lines of communication increase. New developers spend more time comprehending the existing system.
Technical debt	Increased complexity in a system – including debt from previous build decisions – increase the difficulty of making changes

For example, Hailo wanted to expand internationally – which would have been challenging with their original architecture – but also increase their pace of feature delivery.[3] Soundcloud wanted to be more productive, as the complexity of their original monolithic application was holding them back.[4] 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 with microservices, rather than building a single application first. For example, Monzo started with microservices as part of their mission to build a better and more scalable bank.[5]

But what makes this architecture a good choice? There are plenty of successful businesses built on monolithic software – Basecamp[6], StackOverflow and Etsy spring to mind – and there's a wealth of orthodox software development practice and knowledge fall back on. In some companies, technical heterogeneity might be an obvious reason. For example, at Onfido we started building microservices when we introduced a product driven by machine learning – not a great fit for our original Ruby stack! Nevertheless, it's not always so clear-cut.

Let's consider the nature of complex systems. At the beginning of the chapter, we mentioned that software developers strive to craft elegant 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! What we can do is ensure that the development approaches we take result in *good* complex systems, rather than poor ones. Let's think about what we're trying to achieve as software developers. 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 delivering value *sustainably* in the face of change; to continue to deliver with agility, pace and safety even as the system becomes larger

[3] https://sudo.hailoapp.com/services/2015/03/09/journey-into-a-microservice-world-part-1/
[4] http://philcalcado.com/2015/09/08/how_we_ended_up_with_microservices.html
[5] https://speakerdeck.com/mattheath/building-microservice-architectures-in-go
[6] David Heinemeier Hansson coined the term "Majestic Monolith" to describe how 37Signals built Basecamp: https://m.signalvnoise.com/the-majestic-monolith-29166d022228#.6674fa2ze

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 limits our velocity and agility, and therefore our 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: CI jobs, releases – even local application start-up – become slower.

We can't claim these qualities are 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 above.

So how can microservices help? Isolating and minimizing dependencies at build time – whether between teams or on existing code – allows developers to move faster. Table 1.3 compares microservices to monolithic applications regarding friction.

Table 1.3 Comparing friction between microservice and monolithic applications

Pressure	Monolith	Microservices
Component Boundaries	Diffuse	Clear
Ownership Boundaries	Diffuse	Clear
Replacing components	Hard	Easy
Development (Teams)	Coupled; Sequential; Potentially impacting other teams	Independent; Parallel; Not impacting other teams
Technical debt	Potentially spread across the whole application	Encapsulated per service

You can see that microservices are easy to build and reason about *individually.* This is beneficial both for the productivity of development in a growing organization, but also provides a compelling and flexible paradigm for coping with increased scale or smoothly introducing new technologies.

The small size of microservices is also a great enabler of continuous delivery. Deploys in large applications can be risky and involve lengthy change, regression and verification cycles. By deploying smaller elements of functionality, we better isolate changes to our active system, reducing the risk of each individual deploy.

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, we can build a system

that is malleable and resilient in the face of change, helping us to sustainably deliver business impact

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 we can only say "it depends" – on your team, on your company, and on the nature of the system that you're building. If the scope of your system is trivial, then it's unlikely that 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 could be a compelling solution.

Next, let's dig a little deeper and explore the costs and complexity of designing and running microservices.

1.2 What makes microservices challenging?

Microservices aren't the only architecture that has promised nirvana through decomposition and distribution – but those past attempts, such as SOA[7], are widely considered unsuccessful. Unfortunately, no technique is a silver bullet. As we've mentioned, microservices drastically increase moving parts in a system. By distributing functionality and data ownership across multiple autonomous services, we likewise distribute responsibility for stability and sane operation of our application.

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

- *Scoping and identifying microservices* requires substantial domain knowledge
- The right *boundaries and contracts* between services are difficult to identify, and once established, 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

Let's look at how these challenges impact the design and runtime phases of microservice development.

1.2.1 Design challenges

Earlier we introduced the five key principles underlying microservice development. The first of those was *autonomy*. For our services to be autonomous, we need to design them such that

[7] **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. A good overview of the differences between the two approaches is available here: http://oreil.ly/2ctDv4F**

together, they're loosely coupled, and individually, encapsulate highly cohesive elements of functionality. This is an evolutionary process: the scope of our services may change over time, and we'll often choose to carve out new functionality from, or even retire, existing services.

But 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. Especially early in an application's lifetime, your domain knowledge might be at best incomplete, or at worst, incorrect.

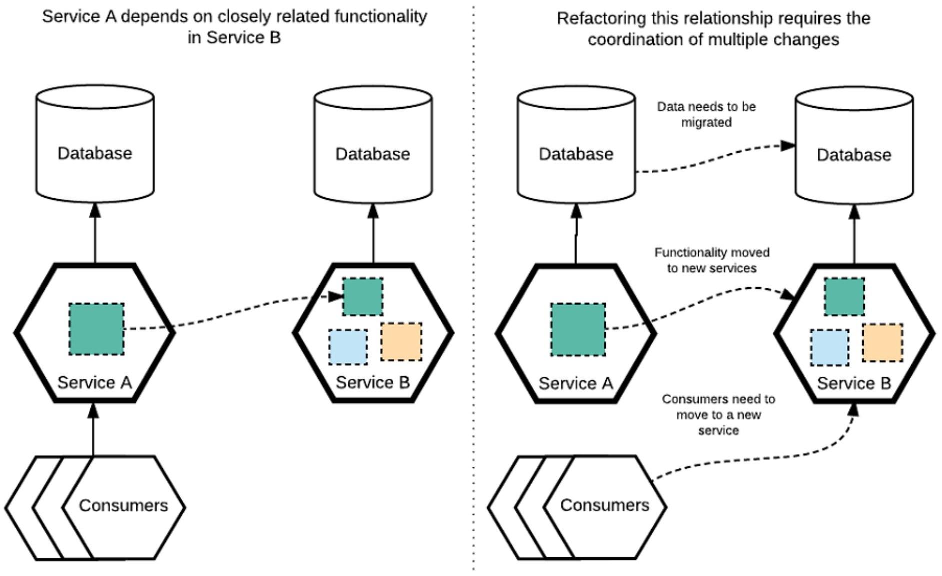
Service A depends on closely related functionality in Service B
Refactoring this relationship requires the coordination of multiple changes
Database
Database
Database
Data needs to be migrated
Database
Functionality moved to new services
Service A
Service B
Service A
Service B
Consumers need to move to a new service
Consumers
Consumers

Figure 1.4 Incorrect service scoping decisions may require complex and costly refactoring across service boundaries.

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

- Refactoring needs to take place across multiple distinct codebases
- Data may need to be migrated from one service's database to another
- Implicit dependencies between services may not be identified, leading to errors or

incompatibility upon deployment

These activities are illustrated in Figure 1.4. Of course, 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*: define the full scope of an interaction
- *Succinct*: take in no more information than is necessary, such that consumers can construct messages within reasonable bounds
- *Predictable*: accurately reflect the real behaviour 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, but need 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 organisations, it's very likely that a microservice application will be built and run by multiple teams, each taking responsibility for different microservices. Each team may have its own goals, their own way of working, and their own 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 will therefore require the agreement and reconciliation of priorities and practices across multiple teams in order to achieve overall business goals.

MICROSERVICE APPLICATIONS ARE DISTRIBUTED SYSTEMS

Lastly, designing microservices applications means designing distributed systems. There are many fallacies made in the design of distributed systems,[8] including:

[8] □ https://pages.cs.wisc.edu/~zuyu/files/fallacies.pdf

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

It's clear that assumptions that we might make in non-distributed systems – such as the speed and reliability of method calls – are no longer appropriate and can lead to poor, unstable implementation. Instead, you will need to consider latency, reliability and the consistency of state across your application.

1.2.2 Operational challenges

A microservices approach will inherently multiply the possible points of failure in a system. To illustrate this, let's return to our investment tool. Figure 1.5 identifies possible points of failure in this application. You can see that there are multiple places where something could go wrong and affect sane processing of an order.

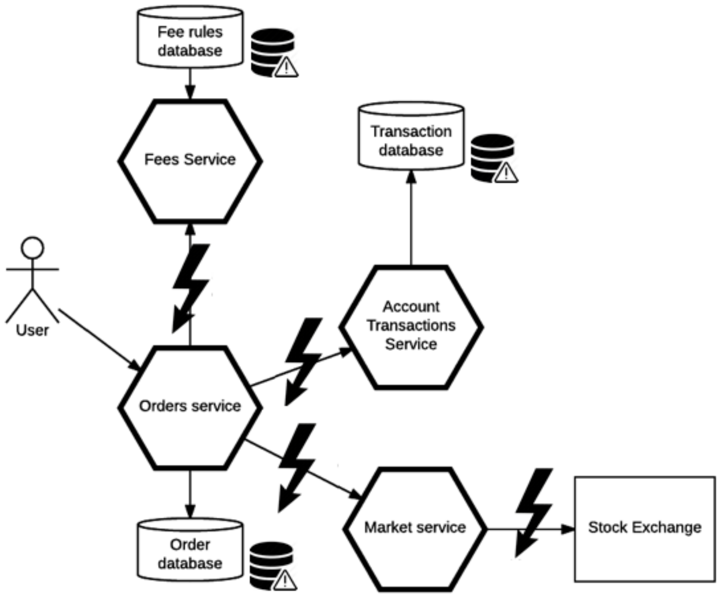

Figure 1.5 Possible points of failure when placing a sell order include database and network

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

- If something goes wrong and our user's order isn't placed, how would we determine where the fault occurred?
- How do we deploy a new version of a service without affecting order placement?
- How do we know which services were meant to be called?

- How do we test this behavior is working correctly across multiple services?
- What happens if a service is unavailable?

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

A microservices approach suggests an evolutionary approach to system design: ideally new features can be added independently, without change to existing services. This minimizes the cost and risk of change. But a decoupled and continuously changing system is also more difficult to understand. Let's build a new feature for our investment tool (Figure 1.6). At our bank, we segment customers into tiers – gold, silver and bronze – based on how many orders they place a month:

1. When an order is placed, the *Order* service triggers an `OrderCreated` event
2. The *Tier* service subscribes to this event
3. When an event is received, the *Tier* service checks the qualification of a customer by summing up the number of order events received within the current evaluation period
4. If a customer qualifies (or is disqualified), the *Tier* service records the tier and triggers a `QualificationChanged` event.
5. Other services subscribe to that event; for example, the *Fees* service might record the qualification of a customer to determine whether to charge a higher or lower fee on an order

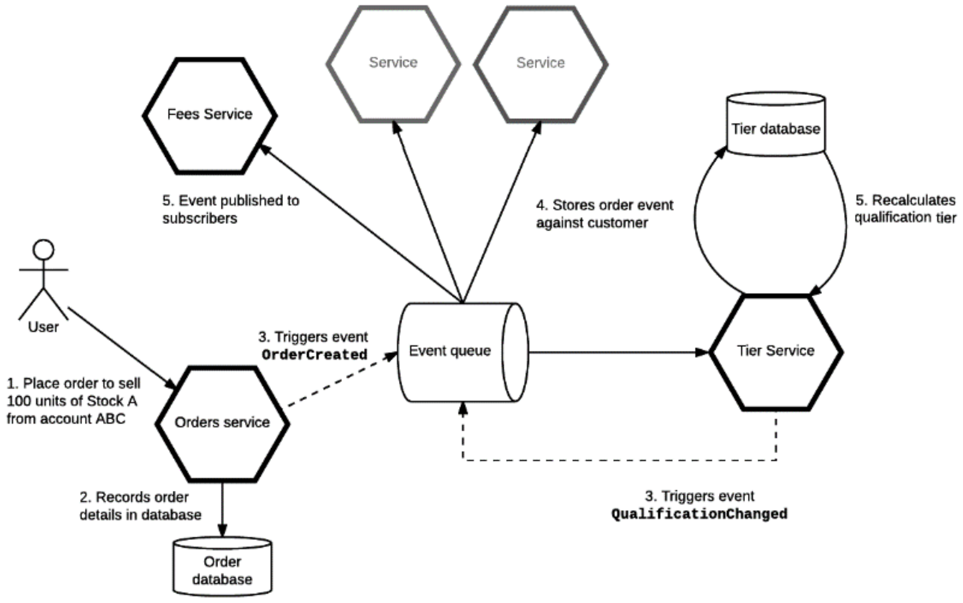
Service
Service
Fees Service
Tier database
5. Event published to subscribers
4. Stores order event against customer
5. Recalculates qualification tier
User
3. Triggers event OrderCreated
Event queue
Tier Service
1. Place order to sell 100 units of Stock A from account ABC
Orders service
2. Records order details in database
3. Triggers event QualificationChanged
Order database

Figure 1.6 Services interact to qualify or disqualify a customer for a reward tier

Imagine a bug is logged: the wrong fee was charged to a customer! To work out what happened, we first need some way of tracing how the system *did* behave (what services where called, in which order, and what was the outcome), but we also need some way of knowing how the system *should* have behaved.

The above are both relatively simple examples. It's not a leap to suspect that this could get harder: imagine you're deploying the market service in New York, Hong Kong, Tokyo and London to minimize latency between the service and local stock exchanges. You might even scale up and scale down capacity when the market opens and closes. It's awesome that microservices let us do this – rather than a cookie-cutter deploy of a single application – but running that kind of application clearly isn't straightforward!

Ultimately, we face two different 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 because we need to understand the big picture. That big picture needs to be assembled from multiple jigsaw pieces: the data each service produces needs to be correlated and linked together to ensure we understand what each service does within the wider context of delivering some business output. Individual service logs provide a partial view of system operation: we need to use both microscope and wide-angle lens to understand the system in full.

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

Therefore, to have a transparent and observable system, we need to collect information from multiple data sources and correlate that in some useful way. Ideally this information can be observed both live – at the point of execution – and historical. But we also need to figure out how to compare that to our expectations and assumptions about how our services should operate.

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 we *start* with that mindset: if we assume weakness and fragility in the constituent parts of our system, then that can better inform how we design, deploy and monitor that system – rather than getting too surprised when something does go wrong!

Therefore, we need to consider how our 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.

Thankfully, this is also an opportunity: we may be able to retain partial service in the face of isolated failure.

NOTE In Chapter 6 we'll teach you how to design resilient and reliable services.

1.3 Microservice development lifecycle

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

But 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, releasing them to production, and observing their behavior. This lifecycle is illustrated in Figure 1.7.

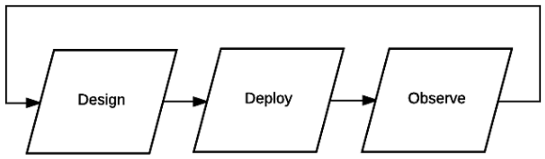

Figure 1.7 The key iterative stages – design, deploy and observe – in the microservice development lifecycle.

Making well-reasoned decisions in each of these three stages will help you build applications that are resilient yet flexible, 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 MVC. But techniques for designing microservices are still in their relative infancy. You'll need to consider:

- Whether to start with a monolith first 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

- How to design services that rely on eventually consistent data

That's quite a lot of ground to cover! For now, let's touch on each of the above so you can see why each of these considerations is vital to a well-designed microservice application.

MONOLITH-FIRST?

There are two opposing trends to starting with microservices: monolith-first or microservices-only. Advocates of the former reason that you should always start by the monolith, as you won't understand the component boundaries in your system at an early stage, as the cost of getting these wrong is much higher in a microservices application.

On the other hand, the boundaries you might choose in a monolith are not necessarily the same as you'd choose in a well-designed microservice application. Although velocity 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.

Nevertheless, 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.

This isn't straightforward! Consider our example from the beginning of this chapter: how might our services change if we wanted to introduce a new, special type of orders? We'd have three options to solve this problem (Figure 1.8):

- Extend the existing service interface
- Add a new service endpoint
- Add a new service

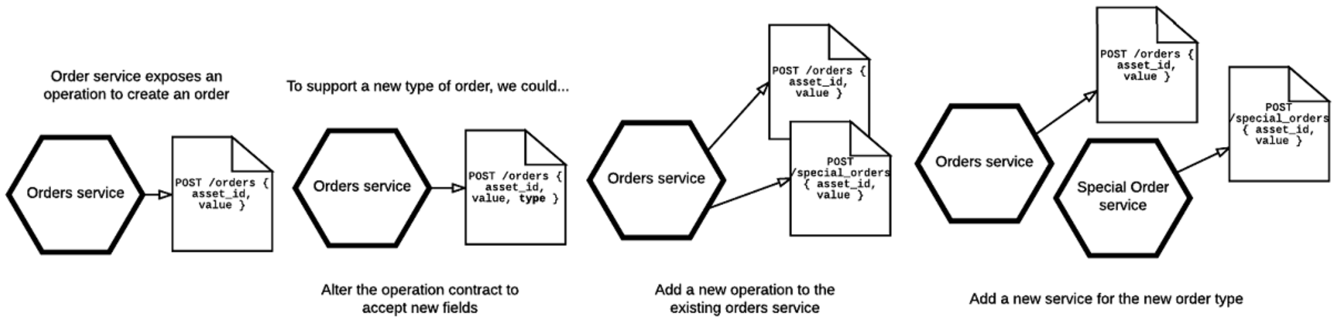
Order service exposes an operation to create an order
Orders service
POST /orders { asset_id, value }
To support a new type of order, we could...
Orders service
POST /orders { asset_id, value, type }
Alter the operation contract to accept new fields
Orders service
POST /orders { asset_id, value }
POST /special_orders { asset_id, value }
Add a new operation to the existing orders service
Orders service
POST /orders { asset_id, value }
Special Order service
POST /special_orders { asset_id, value }
Add a new service for the new order type

Figure 1.8. Scoping functionality involves making decisions about whether capabilities belong in existing services or require new services to be designed.

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

> **NOTE** In Chapter 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. While synchronous systems are easier to reason about, asynchronous systems are highly decoupled – reducing the risk of change – and potentially more resilient. However, the complexity of such a system is high. In a microservices application, you need to balance synchronous and asynchronous messaging to effectively choreograph and coordinate the actions of multiple microservices.

RESILIENCY

In a distributed system, a service can't trust it's collaborators. Not necessarily because they're coded poorly, or because of human error – but because we can't safely assume the network between or behavior of those services is reliable or predictable.

This means that you'll need to design services that work defensively, by backing off in the event of errors, limiting request rates from poor collaborators, and dynamically finding healthy services.

CONSISTENCY

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

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 to be deployed and operated by someone else! 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 our application is the business impact it delivers. That emerges from collaboration between multiple services. In fact, anything outside of the unique capability each service offers could be standardized or abstracted away – ensuring teams are focused on business value. Ultimately, we 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, this process should be standardized to simplify the system operation and ensure consistency across different services. To achieve this, you'll need to:

- Standardize microservice deployment artefacts
- Implement continuous delivery pipelines
- Seamlessly connect different services

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 three steps lead to stable and reliable microservice deploys.

STANDARDIZE MICROSERVICE DEPLOYMENT ARTEFACTS

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 we actually start that application? At runtime, an application 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).

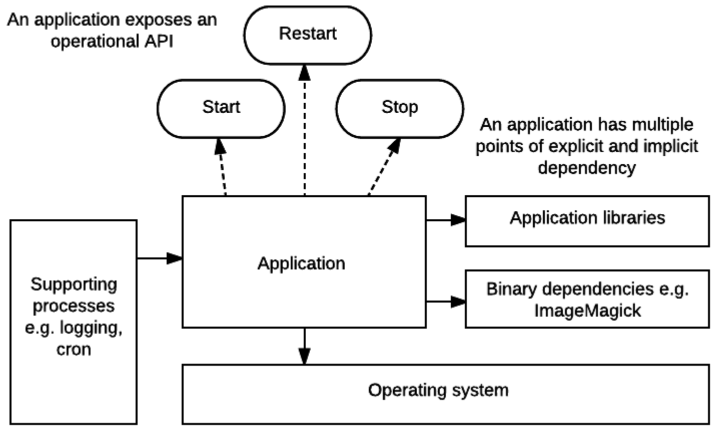
An application exposes an operational API
Restart
Start
Stop
An application has multiple points of explicit and implicit dependency
Application libraries
Application
Supporting processes e.g. logging, cron
Binary dependencies e.g. ImageMagick
Operating system

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

Technically heterogeneity is a fantastic benefit of service autonomy. However, it doesn't make life easy for deployment. Without consistency, we won't be able to standardize our approach to taking services to production, which increases the cost of managing deploys 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.

An ideal deployment artefact for a microservice would allow us to package up a specific version of our compiled code, specifying any binary dependencies, and provide a standard operational abstraction for starting and stopping that service. This should be environment-agnostic: we should be able to run the same artefact locally, in test and in production. By abstracting out differences between languages at runtime, we reduce both cognitive load and provide common abstractions for managing those services.

Our experience suggests the best tool 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, and sharing the same kernel. Unlike a virtual machine, a container is quicker to build and quicker to start up (seconds, rather than minutes). Multiple containers can be run on one machine, which simplifies both local development and can help to optimize resource usage in cloud environments.

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

Although there are many implementations of containers (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 where developers produce software that can be reliably released to production at any time. Imagine a factory production line: to continuously deliver software, we build similar pipelines to take our 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.

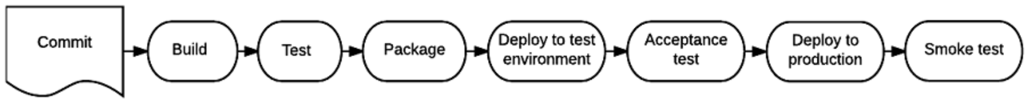

Figure 1.10 A deployment pipeline for a microservice

Earlier, we mentioned that microservices are an ideal enabler of continuous delivery, as their smaller size means that they can be developed quickly and released independently. However, continuous delivery doesn't automatically follow from developing microservices. To continuously deliver software, we need to focus on two goals:

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

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 of delivering new features. Ideally we should be able to standardize the validations and steps in our 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 we can work in a *leaner* fashion – rapidly validating our 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*.**

SEAMLESSLY CONNECT DIFFERENT SERVICES

Once we release a service, it's not necessarily much use by itself. Our services need to interact and collaborate to perform some useful function. We need to be sure that this is reliable: that services can easily find point-to-point collaborators and can register with appropriate queues to receive event-driven messages. For point-to-point collaborators, load must be balanced across multiple instances. If instances fail – and they will – collaborators need to be redirected to healthy services.

This is known more generally as *service discovery*. Within an ecosystem of microservices, this can be especially challenging given the potentially large number of services and the dynamic nature of service collaboration – services that work together today may work together in different ways in the future.

NOTE **In Chapter 12, we'll explore different approaches to service discovery, such as server-side and client-side discovery, and the role of service registries.**

1.3.3 Observing microservices

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

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

Thorough monitoring is significantly more difficult in a microservices application, as single transactions may span multiple distinct services; technically heterogeneous services might product data in irreconcilable formats; and the total volume of operational data is likely to be much higher than a single monolithic application. Luckily, if you understand how your system operates – and observe that closely – then you'll be better placed to make effective changes to that system.

IDENTIFY AND REFACTOR POTENTIALLY FRAGILE IMPLEMENTATION

Systems will fail, whether due to bugs introduced; runtime errors; network failures or hardware problems[9]. 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 us to diagnose problems and determine the causes for failures. We may have automated mechanisms reacting to the alerts that will 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 the cascading of those throughout the system we need to be able to identify and reduce all dependencies between services and architect them in ways that will allow for partial degradation. One service going down should not bring down the whole application.

It is important to think about the possible failure points of our applications, recognize that failure will always happen and prepare accordingly.

UNDERSTAND BEHAVIOUR ACROSS 10S OR 1,000S OF SERVICES

To understand behavior across our services, we need to prioritize transparency in design and implementation. Collecting logs and metrics – and unifying these for analytical and alerting purposes – allows us to build a single source of truth to resort to when monitoring and investigating the behavior of our system.

As we mentioned in Section 1.3.2, anything outside of the unique capability each service offers can be standardized and abstracted. You can think of each service as an onion. At the center of that onion, we have the unique business capability offered by that service. Surrounding that, we have layers of instrumentation – business metrics, application logs, operational metrics and infrastructure metrics – that make that capability observable. Each request to the system can then be traced through these layers. The data collected from these layers would then be pushed to an operational data store for analytics and alerting. This is illustrated in Figure 1.11.

[9] **You've even got to watch out for squirrels: http://www.datacenterknowledge.com/archives/2012/07/09/outages-surviving-electric-squirrels-ups-failures!**

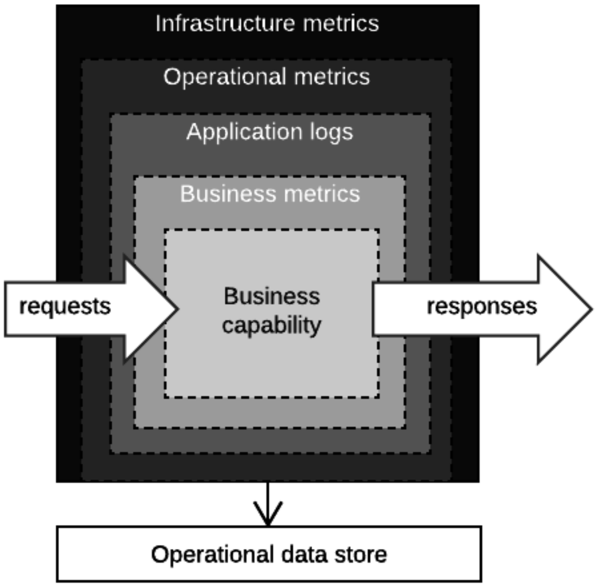

Figure 1.11 A microservice should provide multiple layers of observable data, regardless of the business capability it serves. This data should be considered into an operational data store.

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

1.4 Responsible and "operationally aware" engineering culture

It would 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 is not just an accumulation of features requested, designed and built. It also reflects the preferences, opinions and objectives of its builders and operators.

This is expressed to some degree by Conway's Law:

> **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 might 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 just one example of common microservices *practice*. To be able to realize benefits from microservices and adequately manage their complexity, we need to develop working principles and practices that are effective for that type of application, rather than using the same techniques that we used to build monoliths.

1.5 Summary

This chapter introduced microservice applications, discussed the challenges of designing and running them well, and gave an overview of the design, deployment and monitoring practices we'll focus on throughout this book. In this chapter, you learned that:

- 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, 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 value can be delivered sustainably in these systems if we 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

Teams adopting microservices need to be operationally mature and focus on the entire lifecycle of a service, not just at the design and build stages