Building Microservices

Distributed systems have become more fine-grained in the past 10 years, shifting from code-heavy monolithic applications to smaller, self-contained microservices. But developing these systems brings its own set of headaches. With lots of examples and practical advice, this book takes a holistic view of the topics that system architects and administrators must consider when building, managing, and evolving microservice architectures.

Microservice technologies are moving quickly. Author Sam Newman provides you with a firm grounding in the concepts while diving into current solutions for modeling, integrating, testing, deploying, and monitoring your own autonomous services. You’ll follow a fictional company throughout the book to learn how building a microservice architecture affects a single domain.

- Discover how microservices allow you to align your system design with your organization’s goals
- Learn options for integrating a service with the rest of your system
- Take an incremental approach when splitting monolithic codebases
- Deploy individual microservices through continuous integration
- Examine the complexities of testing and monitoring distributed services
- Manage security with user-to-service and service-to-service models
- Understand the challenges of scaling microservice architectures

Sam Newman is a technologist at ThoughtWorks, where he splits his time between helping clients globally and working as an architect for ThoughtWorks’ internal systems. He has worked with a variety of companies around the world on both development and IT operations.

“The Microservices architecture has many appealing qualities, but the road towards it has painful traps for the unwary. This book will help you figure out if this path is for you, and how to avoid those traps on your journey.”

—Martin Fowler
Chief Scientist, ThoughtWorks

ISBN: 978-1-491-95035-7
US $49.99  CAN $57.99

Twitter: @oreillyphotography
facebook.com/oreilly
Distributed systems have become more fine-grained in the past 10 years, shifting from code-heavy monolithic applications to smaller, self-contained microservices. But developing these systems brings its own set of headaches. With lots of examples and practical advice, this book takes a holistic view of the topics that system architects and administrators must consider when building, managing, and evolving microservice architectures.

Microservice technologies are moving quickly. Author Sam Newman provides you with a firm grounding in the concepts while diving into current solutions for modeling, integrating, testing, deploying, and monitoring your own autonomous services. You’ll follow a fictional company throughout the book to learn how building a microservice architecture affects a single domain.

- Discover how microservices allow you to align your system design with your organization’s goals
- Learn options for integrating a service with the rest of your system
- Take an incremental approach when splitting monolithic codebases
- Deploy individual microservices through continuous integration
- Examine the complexities of testing and monitoring distributed services
- Manage security with user-to-service and service-to-service models
- Understand the challenges of scaling microservice architectures

Sam Newman is a technologist at ThoughtWorks, where he splits his time between helping clients globally and working as an architect for ThoughtWorks’ internal systems. He has worked with a variety of companies around the world on both development and IT operations.
Want to read more?

You can buy this book at oreilly.com in print and ebook format.

Buy 2 books, get the 3rd FREE!
Use discount code: OPC10
All orders over $29.95 qualify for free shipping within the US.

It’s also available at your favorite book retailer, including the iBookstore, the Android Marketplace, and Amazon.com.
# Table of Contents

Preface. ......................................................... xiii

1. **Microservices.** ........................................... 1
   - What Are Microservices? 2
     - Small, and Focused on Doing One Thing Well 2
     - Autonomous 3
   - Key Benefits 4
     - Technology Heterogeneity 4
     - Resilience 5
     - Scaling 5
     - Ease of Deployment 6
     - Organizational Alignment 7
     - Composability 7
     - Optimizing for Replaceability 7
   - What About Service-Oriented Architecture? 8
   - Other Decompositional Techniques 9
     - Shared Libraries 9
     - Modules 10
   - No Silver Bullet 11
   - Summary 11

2. **The Evolutionary Architect.** ......................... 13
   - Inaccurate Comparisons 13
   - An Evolutionary Vision for the Architect 15
   - Zoning 16
   - A Principled Approach 17
     - Strategic Goals 18
     - Principles 18
# Practices

- Combining Principles and Practices
- A Real-World Example
- The Required Standard
- Monitoring
- Interfaces
- Architectural Safety
- Governance Through Code
  - Exemplars
  - Tailored Service Template
- Technical Debt
- Exception Handling
- Governance and Leading from the Center
- Building a Team
- Summary

## 3. How to Model Services

- Introducing MusicCorp
- What Makes a Good Service?
  - Loose Coupling
  - High Cohesion
- The Bounded Context
  - Shared and Hidden Models
  - Modules and Services
  - Premature Decomposition
- Business Capabilities
- Turtles All the Way Down
- Communication in Terms of Business Concepts
- The Technical Boundary
- Summary

## 4. Integration

- Looking for the Ideal Integration Technology
  - Avoid Breaking Changes
  - Keep Your APIs Technology-Agnostic
  - Make Your Service Simple for Consumers
  - Hide Internal Implementation Detail
- Interfacing with Customers
- The Shared Database
- Synchronous Versus Asynchronous
- Orchestration Versus Choreography
- Remote Procedure Calls
Technology Coupling
Local Calls Are Not Like Remote Calls
Brittleness
Is RPC Terrible?
REST
  REST and HTTP
  Hypermedia As the Engine of Application State
  JSON, XML, or Something Else?
  Beware Too Much Convenience
  Downsides to REST Over HTTP
Implementing Asynchronous Event-Based Collaboration
  Technology Choices
  Complexities of Asynchronous Architectures
Services as State Machines
Reactive Extensions
DRY and the Perils of Code Reuse in a Microservice World
  Client Libraries
Access by Reference
Versioning
  Defer It for as Long as Possible
  Catch Breaking Changes Early
  Use Semantic Versioning
  Coexist Different Endpoints
  Use Multiple Concurrent Service Versions
User Interfaces
  Toward Digital
  Constraints
  API Composition
  UI Fragment Composition
  Backends for Frontends
  A Hybrid Approach
Integrating with Third-Party Software
  Lack of Control
  Customization
  Integration Spaghetti
  On Your Own Terms
  The Strangler Pattern
Summary

5. Splitting the Monolith. ................................................................. 79
It’s All About Seams
Breaking Apart MusicCorp
Defense in Depth: 182
Firewalls: 182
Logging: 182
Intrusion Detection (and Prevention) System: 182
Network Segregation: 183
Operating System: 183
A Worked Example: 184
Be Frugal: 186
The Human Element: 187
The Golden Rule: 187
Baking Security In: 187
External Verification: 188
Summary: 188

10. **Conway’s Law and System Design** ........................................ 191
Evidence: 191
   - Loose and Tightly Coupled Organizations: 192
   - Windows Vista: 192
Netflix and Amazon: 192
What Can We Do with This?: 193
Adapting to Communication Pathways: 193
Service Ownership: 194
Drivers for Shared Services: 195
   - Too Hard to Split: 195
   - Feature Teams: 195
   - Delivery Bottlenecks: 196
Internal Open Source: 196
   - Role of the Custodians: 197
   - Maturity: 197
   - Tooling: 198
Bounded Contexts and Team Structures: 198
The Orphaned Service?: 198
Case Study: RealEstate.com.au: 199
Conway’s Law in Reverse: 201
People: 202
Summary: 203

11. **Microservices at Scale** ..................................................... 205
Failure Is Everywhere: 205
How Much Is Too Much?: 206
Degrading Functionality: 207
Architectural Safety Measures: 208
CHAPTER 1

Microservices

For many years now, we have been finding better ways to build systems. We have been learning from what has come before, adopting new technologies, and observing how a new wave of technology companies operate in different ways to create IT systems that help make both their customers and their own developers happier.

Eric Evans's book *Domain-Driven Design* (Addison-Wesley) helped us understand the importance of representing the real world in our code, and showed us better ways to model our systems. The concept of continuous delivery showed how we can more effectively and efficiently get our software into production, instilling in us the idea that we should treat every check-in as a release candidate. Our understanding of how the Web works has led us to develop better ways of having machines talk to other machines. Alistair Cockburn's concept of hexagonal architecture guided us away from layered architectures where business logic could hide. Virtualization platforms allowed us to provision and resize our machines at will, with infrastructure automation giving us a way to handle these machines at scale. Some large, successful organizations like Amazon and Google espoused the view of small teams owning the full lifecycle of their services. And, more recently, Netflix has shared with us ways of building antifragile systems at a scale that would have been hard to comprehend just 10 years ago.

Domain-driven design. Continuous delivery. On-demand virtualization. Infrastructure automation. Small autonomous teams. Systems at scale. Microservices have emerged from this world. They weren't invented or described before the fact; they emerged as a trend, or a pattern, from real-world use. But they exist only because of all that has gone before. Throughout this book, I will pull strands out of this prior work to help paint a picture of how to build, manage, and evolve microservices.

Many organizations have found that by embracing fine-grained, microservice architectures, they can deliver software faster and embrace newer technologies. Microser-
vices give us significantly more freedom to react and make different decisions, allowing us to respond faster to the inevitable change that impacts all of us.

**What Are Microservices?**

Microservices are small, autonomous services that work together. Let’s break that definition down a bit and consider the characteristics that make microservices different.

**Small, and Focused on Doing One Thing Well**

Codebases grow as we write code to add new features. Over time, it can be difficult to know where a change needs to be made because the codebase is so large. Despite a drive for clear, modular monolithic codebases, all too often these arbitrary in-process boundaries break down. Code related to similar functions starts to become spread all over, making fixing bugs or implementations more difficult.

Within a monolithic system, we fight against these forces by trying to ensure our code is more cohesive, often by creating abstractions or modules. Cohesion—the drive to have related code grouped together—is an important concept when we think about microservices. This is reinforced by Robert C. Martin’s definition of the *Single Responsibility Principle*, which states “Gather together those things that change for the same reason, and separate those things that change for different reasons.”

Microservices take this same approach to independent services. We focus our service boundaries on business boundaries, making it obvious where code lives for a given piece of functionality. And by keeping this service focused on an explicit boundary, we avoid the temptation for it to grow too large, with all the associated difficulties that this can introduce.

The question I am often asked is *how small is small?* Giving a number for lines of code is problematic, as some languages are more expressive than others and can therefore do more in fewer lines of code. We must also consider the fact that we could be pulling in multiple dependencies, which themselves contain many lines of code. In addition, some part of your domain may be legitimately complex, requiring more code. Jon Eaves at RealEstate.com.au in Australia characterizes a microservice as something that could be rewritten in two weeks, a rule of thumb that makes sense for his particular context.

Another somewhat trite answer I can give is *small enough and no smaller.* When speaking at conferences, I nearly always ask the question *who has a system that is too big and that you’d like to break down?* Nearly everyone raises their hands. We seem to have a very good sense of what is too big, and so it could be argued that once a piece of code no longer *feels* too big, it’s probably small enough.
A strong factor in helping us answer *how small?* is how well the service aligns to team structures. If the codebase is too big to be managed by a small team, looking to break it down is very sensible. We’ll talk more about organizational alignment later on.

When it comes to how small is small enough, I like to think in these terms: the smaller the service, the more you maximize the benefits and downsides of microservice architecture. As you get smaller, the benefits around interdependence increase. But so too does some of the complexity that emerges from having more and more moving parts, something that we will explore throughout this book. As you get better at handling this complexity, you can strive for smaller and smaller services.

**Autonomous**

Our microservice is a separate entity. It might be deployed as an isolated service on a platform as a service (PAAS), or it might be its own operating system process. We try to avoid packing multiple services onto the same machine, although the definition of *machine* in today’s world is pretty hazy! As we’ll discuss later, although this isolation can add some overhead, the resulting simplicity makes our distributed system much easier to reason about, and newer technologies are able to mitigate many of the challenges associated with this form of deployment.

All communication between the services themselves are via network calls, to enforce separation between the services and avoid the perils of tight coupling.

These services need to be able to change independently of each other, and be deployed by themselves without requiring consumers to change. We need to think about what our services should expose, and what they should allow to be hidden. If there is too much sharing, our consuming services become coupled to our internal representations. This decreases our autonomy, as it requires additional coordination with consumers when making changes.

Our service exposes an application programming interface (API), and collaborating services communicate with us via those APIs. We also need to think about what technology is appropriate to ensure that this itself doesn’t couple consumers. This may mean picking technology-agnostic APIs to ensure that we don’t constrain technology choices. We’ll come back time and again to the importance of good, decoupled APIs throughout this book.

Without decoupling, everything breaks down for us. The golden rule: can you make a change to a service and deploy it by itself without changing anything else? If the answer is no, then many of the advantages we discuss throughout this book will be hard for you to achieve.

To do decoupling well, you’ll need to model your services right and get the APIs right. I’ll be talking about that a lot.
Key Benefits

The benefits of microservices are many and varied. Many of these benefits can be laid at the door of any distributed system. Microservices, however, tend to achieve these benefits to a greater degree primarily due to how far they take the concepts behind distributed systems and service-oriented architecture.

Technology Heterogeneity

With a system composed of multiple, collaborating services, we can decide to use different technologies inside each one. This allows us to pick the right tool for each job, rather than having to select a more standardized, one-size-fits-all approach that often ends up being the lowest common denominator.

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

![Figure 1-1. Microservices can allow you to more easily embrace different technologies](image)

With microservices, we are also able to adopt technology more quickly, and understand how new advancements may help us. One of the biggest barriers to trying out and adopting new technology is the risks associated with it. With a monolithic application, if I want to try a new programming language, database, or framework, any change will impact a large amount of my system. With a system consisting of multiple services, I have multiple new places in which to try out a new piece of technology. I can pick a service that is perhaps lowest risk and use the technology there, knowing that I can limit any potential negative impact. Many organizations find this ability to more quickly absorb new technologies to be a real advantage for them.
Embracing multiple technologies doesn’t come without an overhead, of course. Some organizations choose to place some constraints on language choices. Netflix and Twitter, for example, mostly use the Java Virtual Machine (JVM) as a platform, as they have a very good understanding of the reliability and performance of that system. They also develop libraries and tooling for the JVM that make operating at scale much easier, but make it more difficult for non-Java-based services or clients. But neither Twitter nor Netflix use only one technology stack for all jobs, either. Another counterpoint to concerns about mixing in different technologies is the size. If I really can rewrite my microservice in two weeks, you may well mitigate the risks of embracing new technology.

As you’ll find throughout this book, just like many things concerning microservices, it’s all about finding the right balance. We’ll discuss how to make technology choices in Chapter 2, which focuses on evolutionary architecture; and in Chapter 4, which deals with integration, you’ll learn how to ensure that your services can evolve their technology independently of each other without undue coupling.

**Resilience**

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

We do need to be careful, however. To ensure our microservice systems can properly embrace this improved resilience, we need to understand the new sources of failure that distributed systems have to deal with. Networks can and will fail, as will machines. We need to know how to handle this, and what impact (if any) it should have on the end user of our software.

We’ll talk more about better handling resilience, and how to handle failure modes, in Chapter 11.

**Scaling**

With a large, monolithic service, we have to scale everything together. One small part of our overall system is constrained in performance, but if that behavior is locked up in a giant monolithic application, we have to handle scaling everything as a piece. With smaller services, we can just scale those services that need scaling, allowing us to run other parts of the system on smaller, less powerful hardware, like in Figure 1-2.
Gilt, an online fashion retailer, adopted microservices for this exact reason. Starting in 2007 with a monolithic Rails application, by 2009 Gilt’s system was unable to cope with the load being placed on it. By splitting out core parts of its system, Gilt was better able to deal with its traffic spikes, and today has over 450 microservices, each one running on multiple separate machines.

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

**Ease of Deployment**

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

With microservices, we can make a change to a single service and deploy it independently of the rest of the system. This allows us to get our code deployed faster. If a problem does occur, it can be isolated quickly to an individual service, making fast rollback easy to achieve. It also means we can get our new functionality out to customers faster. This is one of the main reasons why organizations like Amazon and Netflix use these architectures—to ensure they remove as many impediments as possible to getting software out the door.
The technology in this space has changed greatly in the last couple of years, and we'll be looking more deeply into the topic of deployment in a microservice world in Chapter 6.

**Organizational Alignment**

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

Microservices allow us to better align our architecture to our organization, helping us minimize the number of people working on any one codebase to hit the sweet spot of team size and productivity. We can also shift ownership of services between teams to try to keep people working on one service colocated. We will go into much more detail on this topic when we discuss Conway’s law in Chapter 10.

**Composability**

One of the key promises of distributed systems and service-oriented architectures is that we open up opportunities for reuse of functionality. With microservices, we allow for our functionality to be consumed in different ways for different purposes. This can be especially important when we think about how our consumers use our software. Gone is the time when we could think narrowly about either our desktop website or mobile application. Now we need to think of the myriad ways that we might want to weave together capabilities for the Web, native application, mobile web, tablet app, or wearable device. As organizations move away from thinking in terms of narrow channels to more holistic concepts of customer engagement, we need architectures that can keep up.

With microservices, think of us opening up seams in our system that are addressable by outside parties. As circumstances change, we can build things in different ways. With a monolithic application, I often have one coarse-grained seam that can be used from the outside. If I want to break that up to get something more useful, I’ll need a hammer! In Chapter 5, I’ll discuss ways for you to break apart existing monolithic systems, and hopefully change them into some reusable, re-composable microservices.

**Optimizing for Replaceability**

If you work at a medium-size or bigger organization, chances are you are aware of some big, nasty legacy system sitting in the corner. The one no one wants to touch. The one that is vital to how your company runs, but that happens to be written in some odd Fortran variant and runs only on hardware that reached end of life 25 years ago. Why hasn’t it been replaced? You know why: it’s too big and risky a job.
With our individual services being small in size, the cost to replace them with a better implementation, or even delete them altogether, is much easier to manage. How often have you deleted more than a hundred lines of code in a single day and not worried too much about it? With microservices often being of similar size, the barriers to rewriting or removing services entirely are very low.

Teams using microservice approaches are comfortable with completely rewriting services when required, and just killing a service when it is no longer needed. When a codebase is just a few hundred lines long, it is difficult for people to become emotionally attached to it, and the cost of replacing it is pretty small.

What About Service-Oriented Architecture?

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

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

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

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

Much of the conventional wisdom around SOA doesn't help you understand how to split something big into something small. It doesn't talk about how big is too big. It doesn't talk enough about real-world, practical ways to ensure that services do not become overly coupled. The number of things that go unsaid is where many of the pitfalls associated with SOA originate.
The microservice approach has emerged from real-world use, taking our better understanding of systems and architecture to do SOA well. So you should instead think of microservices as a specific approach for SOA in the same way that XP or Scrum are specific approaches for Agile software development.

**Other Decompositional Techniques**

When you get down to it, many of the advantages of a microservice-based architecture come from its granular nature and the fact that it gives you many more choices as to how to solve problems. But could similar decompositional techniques achieve the same benefits?

**Shared Libraries**

A very standard decompositional technique that is built into virtually any language is breaking down a codebase into multiple libraries. These libraries may be provided by third parties, or created in your own organization.

Libraries give you a way to share functionality between teams and services. I might create a set of useful collection utilities, for example, or perhaps a statistics library that can be reused.

Teams can organize themselves around these libraries, and the libraries themselves can be reused. But there are some drawbacks.

First, you lose true technology heterogeneity. The library typically has to be in the same language, or at the very least run on the same platform. Second, the ease with which you can scale parts of your system independently from each other is curtailed. Next, unless you’re using dynamically linked libraries, you cannot deploy a new library without redeploying the entire process, so your ability to deploy changes in isolation is reduced. And perhaps the kicker is that you lack the obvious seams around which to erect architectural safety measures to ensure system resiliency.

Shared libraries do have their place. You’ll find yourself creating code for common tasks that aren’t specific to your business domain that you want to reuse across the organization, which is an obvious candidate for becoming a reusable library. You do need to be careful, though. Shared code used to communicate between services can become a point of coupling, something we’ll discuss in Chapter 4.

Services can and should make heavy use of third-party libraries to reuse common code. But they don’t get us all the way there.
Modules

Some languages provide their own modular decomposition techniques that go beyond simple libraries. They allow some lifecycle management of the modules, such that they can be deployed into a running process, allowing you to make changes without taking the whole process down.

The Open Source Gateway Initiative (OSGI) is worth calling out as one technology-specific approach to modular decomposition. Java itself doesn't have a true concept of modules, and we'll have to wait at least until Java 9 to see this added to the language. OSGI, which emerged as a framework to allow plug-ins to be installed in the Eclipse Java IDE, is now used as a way to retrofit a module concept in Java via a library.

The problem with OSGI is that it is trying to enforce things like module lifecycle management without enough support in the language itself. This results in more work having to be done by module authors to deliver on proper module isolation. Within a process boundary, it is also much easier to fall into the trap of making modules overly coupled to each other, causing all sorts of problems. My own experience with OSGI, which is matched by that of colleagues in the industry, is that even with good teams it is easy for OSGI to become a much bigger source of complexity than its benefits warrant.

Erlang follows a different approach, in which modules are baked into the language runtime. Thus, Erlang is a very mature approach to modular decomposition. Erlang modules can be stopped, restarted, and upgraded without issue. Erlang even supports running more than one version of the module at a given time, allowing for more graceful module upgrading.

The capabilities of Erlang’s modules are impressive indeed, but even if we are lucky enough to use a platform with these capabilities, we still have the same shortcomings as we do with normal shared libraries. We are strictly limited in our ability to use new technologies, limited in how we can scale independently, can drift toward integration techniques that are overly coupling, and lack seams for architectural safety measures.

There is one final observation worth sharing. Technically, it should be possible to create well-factored, independent modules within a single monolithic process. And yet we rarely see this happen. The modules themselves soon become tightly coupled with the rest of the code, surrendering one of their key benefits. Having a process boundary separation does enforce clean hygiene in this respect (or at least makes it harder to do the wrong thing!). I wouldn’t suggest that this should be the main driver for process separation, of course, but it is interesting that the promises of modular separation within process boundaries rarely deliver in the real world.

So while modular decomposition within a process boundary may be something you want to do as well as decomposing your system into services, by itself it won’t help
solve everything. If you are a pure Erlang shop, the quality of Erlang’s module implementation may get you a very long way, but I suspect many of you are not in that situation. For the rest of us, we should see modules as offering the same sorts of benefits as shared libraries.

**No Silver Bullet**

Before we finish, I should call out that microservices are no free lunch or silver bullet, and make for a bad choice as a golden hammer. They have all the associated complexities of distributed systems, and while we have learned a lot about how to manage distributed systems well (which we’ll discuss throughout the book) it is still hard. If you’re coming from a monolithic system point of view, you’ll have to get much better at handling deployment, testing, and monitoring to unlock the benefits we’ve covered so far. You’ll also need to think differently about how you scale your systems and ensure that they are resilient. Don’t also be surprised if things like distributed transactions or CAP theorem start giving you headaches, either!

Every company, organization, and system is different. A number of factors will play into whether or not microservices are right for you, and how aggressive you can be in adopting them. Throughout each chapter in this book I’ll attempt to give you guidance highlighting the potential pitfalls, which should help you chart a steady path.

**Summary**

Hopefully by now you know what a microservice is, what makes it different from other compositional techniques, and what some of the key advantages are. In each of the following chapters we will go into more detail on how to achieve these benefits and how to avoid some of the common pitfalls.

There are a number of topics to cover, but we need to start somewhere. One of the main challenges that microservices introduce is a shift in the role of those who often guide the evolution of our systems: the architects. We’ll look next at some different approaches to this role that can ensure we get the most out of this new architecture.
When you buy an ebook through oreilly.com you get lifetime access to the book, and whenever possible we provide it to you in five, DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, Android .apk, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we’ve updated the files with corrections and additions.

Learn more at ebooks.oreilly.com

You can also purchase O’Reilly ebooks through the iBookstore, the Android Marketplace, and Amazon.com.