O'REILLY®

Microservices Up & Running

A Step-by-Step Guide to Building a Microservices Architecture

Free Chapters

compliments of

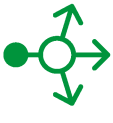
NGINX®
Part of F5

Ronnie Mitra &
Irakli Nadareishvili

Improve the Performance, Reliability, and Security of Your Applications with NGINX

Scale, Deploy, and Protect with Ease

Whether you need to integrate advanced monitoring, strengthen security controls, or manage Kubernetes workloads, there's an NGINX solution for you – from our open source offerings to enterprise-grade products like NGINX Plus, NGINX Controller, and NGINX App Protect. Discover a faster and more reliable way to manage your digital realm – that modern app and DevOps teams will love – with NGINX.



Load Balancing

- Deploy anywhere
- Integrate easily

Microservices

- End-to-end Encryption
- Layer 7 routing

Cloud

- Support unlimited apps
- Get predictable pricing

Security

- DDoS mitigation
- Elliptic Curve Cryptography

Web and Mobile Apps

- Reduce page load time
- Scale when you need it

API Gateway

- API authentication and authorization
- Real-time monitoring and alerting

Download a 30-day free trial today at: nginx.com/free-trial-request/

NGINX
Part of F5

Microservices: Up and Running

A Step-by-Step Guide to Building a Microservices Architecture

This excerpt contains Chapters 2, 4, and 7. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Ronnie Mitra and Irakli Nadareishvili

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

Microservices: Up and Running
by Ronnie Mitra and Irakli Nadareishvili

Printed in the United States of America.

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

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

Acquisitions Editor: Melissa Duffield
Development Editor: Melissa Potter
Production Editor: Deborah Baker
Copyeditor: Charles Roumeliotis
Proofreader: Piper Editorial, LLC

Indexer: nSight, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

December 2020: First Edition

Revision History for the First Edition
2020-11-25: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492075455* for release details.

978-1-492-07545-5

[LSI]

To every person who took the time to chronicle and share their experiences. And to Kairav, who didn't help me write this dedication.

—Ronnie Mitra

To Lucas, who was born shortly after we started working on this book and whose smiles gave me the strength to complete this book in the middle of a global pandemic; to my wife Ana, for her support; and to my amazing students at Temple University, in Philadelphia, who kindly "test drove" early versions of a lot of the content in this book.

—Irakli

Table of Contents

Foreword

It all started with a "Fail Whale." When Twitter's monolithic application began to tip over with alarming regularity, teams inside the breakout social media success desperately sought a way to make the service more reliable. Teams inside of Twitter began to decompose the clunky Rails application into a series of smaller applications linked together via RESTful APIs.

Thus began the microservices revolution. Aided and abetted by the emergence of cloud computing and containerized applications, digital native and cloud-first companies quickly shifted away from legacy application topologies—giant, monolithic bodies of code—to more nimble microservice architectures. Twitter, Netflix, Uber, and Facebook, to name a few, all created platforms and compound applications from smaller, simpler microservices.

Today, microservices are positively mainstream. Entire tracks at many major conferences are focused on illuminating the mysteries of microservices and describing how companies today are building platforms to support microservices. That said, adopting microservices is not just like flipping a switch or buying some cloud instances. Building microservice-based applications requires deep cooperation between teams, new ways of thinking about security, and new approaches to integrating and deploying code. For this reason, to successfully embrace microservice applications and architectures, CTOs and their teams need to embrace an entirely new and different way of thinking about building software products.

In this thorough book, Ronnie Mitra and Irakli Nadareishvili offer among the most detailed explanations and playbooks yet for microservice architectures and successfully building out microservice applications. Written for implementers and builders, the book details each stage of the journey from planning and consideration to deployment to ongoing operations and code change management. The book is written in accessible terms that make sense to engineers but are accessible to nontechnical project managers.

It is also clear from the book that the authors have spent time in the trenches building and operating microservice-driven applications—years, in fact. (This is their second book on the topic.) Their experience shines through in numerous detail-oriented explanations covering not only the mechanics of microservices but also the cultural shifts and the necessary processes for driving lasting adoption. The result is a rich holistic perspective on what it really means to make microservices a guiding design principle.

For organizations and engineers that wish to learn about microservices, this book can serve as a quick start manual and broad overview of this rich vein. For those wishing to actually implement microservices, the book works equally as well as a detailed project roadmap and adoption guide. For companies that are already building complex microservice architectures, this book can provide useful ideas and insights into how to make their processes and applications more efficient and more effective.

The truth is this: we are not going back to the world of monolithic applications. Modern web-scale and mobile applications require the agility and resilience of microservices. The entire architecture of applications is now widely distributed in a manner that meshes perfectly with the distributed designs of microservices. Equally important, a host of technologies—like service meshes, Kubernetes, and containers—have made deploying and scaling microservices easier and more economical. If engineers were to rebuild Twitter today, they would have chosen microservice architectures from the very start.

With this in mind, creating an effective microservices strategy and gaining expertise in how to build and scale them is table stakes for modern technology organizations. The authors of this book deliver one of the fastest and most comprehensive on-ramps to the discipline. By building on many existing useful concepts for application development and adding their own twists from hard-won insights of years architecting and managing distributed applications, they set up a smooth glidepath to microservices mastery that even novices will feel comfortable attempting.

— Karthik Krishnaswamy
Head of Product Marketing, NGINX

CHAPTER 2

Designing a Microservices Operating Model

In this book, we'll be building a microservices-based application. To do that, we'll design and build microservices as well as the infrastructure and tools you need to support them. However, the truth is that success with microservices takes more than writing code and deploying it. To really succeed, you need to have the right people, the right ways of working, and the right principles in place to make the whole system work. That's why we want to start our journey by designing a general operating model for our application.

An *operating model* is the set of people, processes, and tools that underlies your system. It shapes all the decision making and work that you do when you build software. For example, an operating model can define the responsibilities of teams. It can also define governance over decision making and work.

You can think of the operating model as the "operating system" for your solution. All the work needed to build microservices happens on top of the team structures, processes, and boundaries you define. In practice, operating models can have a big scope and can be very detailed. But for our build, we'll reduce the scope and focus on the most important parts of a microservices system—how the teams are designed and how they work together.

That's what we'll be covering in this chapter: the relationship between teams and microservices implementations. We'll introduce a tool called Team Topologies and by the end of the chapter we'll have a team-based design that we can use as the foundation for the rest of our build.

You don't need to actually assemble the people and teams we've defined in order to follow along with our "up and running" microservices build.

Let's get started by taking a look at why teams and team design are so important in the first place.

Why Teams and People Matter

The model we're using in this book is mostly concerned with technology and tool decisions. But technology alone won't give you the value you need from a microservices system. Technology is important. Good technology choices make it easier for you to do things that may have been prohibitively difficult. At its best, technology opens doors and unlocks new opportunities. However, it's useless on its own.

You can have the world's best tools and platforms, but you'll fail if you don't have the right culture and organization in which to use them. The goal we're trying to reach in our model is to put good technology in the hands of independent, high-functioning teams. So we'll need to start by considering the types of teams and structure that will work best for the model we're going to develop.

In a microservices system, culture and team design matters. In our research for this book and in our own implementation experiences we've learned an important truth: people and process are critical success factors. A microservices implementation is valuable when it gives you the freedom to make changes easily and quickly. In practice, however, change is a byproduct of your organization's decision-making capability. If you can't make quality decisions quickly, you'll have a difficult time getting value from your microservices. It'd be like building a racing car with a very poor engine. No matter how well the car is built, it's never going to run the way it should.

The idea that team design and culture is important isn't a new one. Mel Conway captured the impact of team structure on system design eloquently in his now-famous article, "How Do Committees Invent?" Mel Conway's insightful observations spawned an even more famous paraphrasing of his thesis, called "Conway's Law":

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.
>
> —Attributed to Fred Brooks

Conway tells us that the output of an organization reflects the way its people and teams communicate. For example, consider a microservices team that must consult a centralized team of database experts whenever they need to change a data model. Chances are that the data model and data implementation will also be centralized in the system that gets produced. The system ends up matching the organization and coordination model.

The takeaway from all this is that the people in a microservices system matter. The way that they make decisions, do their work, and communicate with each other has a big impact on the system that gets produced. Generally speaking, there are three

people factors that have the biggest impact on a microservices system: team size, team skills, and interteam coordination. Let's take a closer look at each of them, starting with size.

Team Size

The "micro" in microservices implies that size matters and smaller is best. To be honest, that's a bit of an oversimplification. But the truth remains: buliding smaller deployable services is an important part of succeeding with microservices. It also turns out that the size of the teams building those services matters a lot too.

If you have too many people on a team, they'll need to spend more time communicating with each other. That internal coordination will end up slowing the team down, resulting in slower delivery of changes. If you have too few people, you won't have enough minds and hands to get the work done. "Rightsizing" teams is an important part of your system design. While there isn't a specific size that works for everyone in all situations, a body of experience and studies on team sizes has evolved into accepted practice.

Bill Gore, cofounder of the Gore-Tex company, W. L. Gore, limited the size of company teams to keep them effective. To make that happen, he instituted a built-in size limit: everyone on a team must have a personal relationship with one another. When a team gets so big that its members don't know each other, the unit has grown too large.

Anthropologist Robert Dunbar, in his studies of the social behavior of chimpanzees, observed that the group sizes of chimpanzees correlated to their brain size. By extrapolating these findings to his understanding of the human brain, he established a set of group sizes for people. The Dunbar number states that we can only comfortably maintain 150 stable relationships, based on the size of our brains. Dunbar also determined that humans could keep about 5 intimate, familial relationships and only about 15 trusted friends.

Perhaps most famously, Amazon CEO Jeff Bezos gave us the "two pizza rule". It states that an Amazon team should be small enough that it can be fed with two pizzas. Although the specific details about the size of the pizzas and the appetite of the team members are unclear, a two-pizza team is probably going to land somewhere in the 5 to 15 person range that Dunbar describes and stands a good chance of maintaining the personal relationship heuristic that Gore describes.

All of these stories point to a size limit based on the ability for people to communicate effectively. Our experiences and our research align with this intuitive concept. To keep the rate of change high, we'll need to limit the size of the teams in our system. In our microservices model, we're going to keep the size of teams to somewhere between five to eight people.

Key Decision: Team Size Should Be Limited

The teams that perform work in our system should have no more than eight people each.

Keeping the team size down will help us limit the internal interaction needed. But it will have a knock-on effect. Smaller team sizes usually mean more teams. So, we'll need to be careful in how we design the rest of the system. It's no good to have small teams if they have to spend all their time coordinating with each other. To avoid that, we'll need to enable independent and autonomous work as much as safely possible.

Another side effect of making our teams smaller is that it limits the number of specialists we can have. With less people on the team, we'll need to make sure we have enough talent collectively to deliver a quality output. That's why we'll need to consider how we populate our teams from a skills perspective.

Team Skills

A team can only be as good as its members. If we want high-performing teams, we'll need to pay special attention to the way we decide who gets to be on a team. For example, which roles and specializations will our teams need? How talented and experienced should individual team members be? What is the right mix of skills and experience?

The truth is that these are difficult questions for us to answer universally. That's because people and culture are often the most unique thing about the place where you work. For example, a handful of companies spend a lot of money to have the top 1% of technology talent from around the world working for them. Another company might mostly hire local talent with a focus on career growth and learning on the job from a small number of experts. Good team design in these two companies will probably look quite different.

We want this book to be focused on building a microservices implementation. So, we won't go very deep into organizational and culture design. The good news is that there is a general principle we can adopt that seems to help microservices implementers universally. That's the principle of the cross-functional team.

In a cross-functional team, people with different types of expertise (or functions) work together toward the same goal. That expertise can span both technology and business domains. For example, a cross-functional team could contain UX designers, application developers, product owners, and business analysts.

Cross-functional teams have been around for a long time, dating back to at least the 1950s at the Northwestern Mutual Life Insurance Company.

A big advantage of building a team this way is that you can make better decisions faster. We've already established an upper limit on team size, by limiting membership to eight people. A "rightsized" team with the right people on board can move at high velocity with authority.

But who are the *right people*? When it came to team size, we had anecdotes, experience, and academic studies to draw on. But for team profiles, it's much more difficult to find consistent stories. For example, when we've seen a large cloud vendor work on microservices, they've used four to five experts with cross-domain knowledge, coupled with a single testing expert. Conversely, we've seen consulting companies use a large mix of specialized engineers, product owners, project managers, and testing experts on each team. The talent, experience, and culture of your organization will inform the precise mix of people.

So, rather than dictate exactly which roles you'll need on your teams, we'll make two general decisions for our model. First, teams should be cross-functional. Our experience shows that microservices work better when teams can make good decisions on their own. Cross-functional teams enable that. Second, teams should be comprised of members who directly influence the output. In this way, we'll pick people who we know can add value to the team. We don't need observers on the team or people who are only tangentially related to the work and decisions that are being made.

Key Decision: Principles for Team Membership Should Be Defined

Teams should be cross-functional and consist only of members who can add value to the team's deliverable, service, or product.

With the right size and the right people, we should be able to build effective teams that can get things done. As the number of teams grow, we'll also need to consider how teams coordinate with each other. That's the last team property we need to address.

Interteam Coordination

Building a team with the right size and filling it with the right people will help us create high-performing teams. But it's the communication among teams, rather than inside them, that can really bog down a microservices system. We highlighted the problem of coordination costs in Chapter 1. If we can reduce the amount of coordi-

nation that takes place between teams, our microservices teams will be able to deliver changes faster.

It would be nice if our microservices teams could act completely autonomously and independently. If teams were free to make their own design, development, testing, and deployment decisions, there would be no "organizational friction" to slow things down. In our experience this isn't a practical method of operation.

That's because coordination and collaboration are important for the success of an organization. We might want our microservices teams to act independently, but we also want them to create services that are valuable to customers, users, and the organization. This means communication is required to establish shared goals, communicate change requests, deliver feedback, and resolve problems.

On top of this, when teams operate completely independently, there's less opportunity to share. Microservices teams working independently can pick the right tools for the right job and build highly efficient systems. But that efficiency is localized to the team. Sometimes, that means we lose out on *system-level* efficiency. For example, if all our teams design and build their own cloud-based network architectures, we've lost an opportunity to do that work once and share it.

It's possible to build an organization that enables efficient team independence and autonomy through self-organization. For example, microservices pioneer Fred George has described a method he calls Programmer Anarchy, in which technology teams have full autonomy (and responsibility) to form teams, choose work, and design their own solutions. But in our experience most enterprise organizations would have difficulty pulling this off consistently.

If we go too far towards team independence and autonomy, we'll introduce system-level inefficiencies and misalignment with organizational goals. If we introduce too much coordination, we risk bogging the whole system down and losing the benefits of highly changeable microservices. The challenge is to strike the right balance between independent work and coordinated efforts. That takes some experimentation and continuous tuning of your team design.

Most importantly, optimizing team coordination requires an active design effort. One of the mistakes we've seen practitioners make is to focus solely on the technical architecture. When that happens, the team design forms around the technology that's been created. It's only then that the problems with the coordination model become obvious. By that point, it's often too costly or too difficult to make changes.

To avoid this problem, we'll address team coordination and team design as the first step of our system design process. Some people call this an "inverse Conway maneuver," because the communication structure we design will end up informing the system that gets created. Whatever you want to call it, we've found that starting with a focus on team design and coordination can really help you succeed with your microservices design. In fact, this point is so important that we'll log it as a decision.

Key Decision: When to Design Teams and Coordination Models

Team and coordination design should start before the design of the system architecture or microservices. The team and coordination models must continually be updated and improved for the life of the system.

We'll cover this in the rest of this chapter. First, we'll introduce a useful tool for designing microservices team models called Team Topologies.

Introducing Team Topologies

Since we're going to start our design work with a focus on teams, we'll need a way of cataloging and communicating our decisions. There are plenty of ways of documenting team designs. For our model, we'll use a design tool called Team Topologies.

Team Topologies is a design approach invented by Matthew Skelton and Manuel Pais. We like using it because it provides a formal language for talking about team design, with a special focus on the way teams work with each other.

We won't be using every aspect of the Team Topologies approach in our design work. Instead, we'll be drawing on three elements: team types, team interaction modes, and diagramming. With these parts, we'll be able to build a simple, working design for our microservices teams.

Next, we'll look at different parts of the Team Topology approach, starting with the types of teams we can define.

Team Types

One of the core concepts of Team Topologies is *team types*. These are archetypes or categories that describe the basic nature of a team, from the perspective of its communication with the rest of an organization. There are four team types defined in Team Topologies: stream-aligned, enabling, complicated-subsystem, and platform. Let's take a quick look at each of them:

Stream-Aligned
: A stream-aligned team owns and runs a deliverable piece of work. The key characteristic of this team is a continual delivery of something relevant to the business organization. The stream-aligned team embodies Amazon CTO Werner Vogel's comment on the responsibilities of Amazon teams: "You build it, you run it." Stream-aligned teams don't disband after a release. Instead, they continue to own and implement a "stream" of changes, improvements, and fixes to their business deliverable. For example, microservices teams are usually stream-aligned as they continually release features to the services they own.

Enabling
: An enabling team supports the work of other teams with a consulting engagement model. These teams are usually composed of specialists and subject matter experts who can bridge gaps in expertise or capability. But they can also help individual teams understand the bigger picture of the organization or industry they are operating in. For example, an enabling architecture team can help microservices teams understand emerging technical standards and conventions in the organization.

Complicated-Subsystem
: This type of team works on a domain or on subject matter that is difficult to understand. Or at least, it's difficult enough that there is a lack of available resources in the organization. Some problem areas don't scale well and can't be embedded in every team. For example, tuning software for cryptographic security requires a special kind of expertise and experience. Rather than trying to scale that skill across all teams, most organizations create a complicated-subsystem security team who can engage with individual teams as needed.

Platform
: Like enabling teams, the platform team provides support to the rest of the organization, with one important difference—platform teams deliver a self-service enablement experience to their users. While the enabling and complicated-subsystem teams are limited by the bandwidth of their people, a platform team invests in building supporting tools and processes that can scale easily. This requires more up-front investment and continual maintenance and support. The platform becomes a product, whose users are the rest of the teams in the organization. For example, operations teams can become platform teams when they offer build and release tools to development teams for them to use.

With an understanding of these four team types, we can start communicating how we want our teams to operate. To really communicate our team design, we will need one more part of the model: the ways in which teams interact with each other, which we'll cover next.

Interaction Modes

Our goal in designing teams for the microservices build is to reduce the amount of coordination that needs to happen for work to get done. The Team Topology team types help us identify the basic characteristics of a team. To really understand how and where we can reduce coordination costs, we'll need to articulate the way our teams are coordinating with each other. That's where the Team Topology interaction modes come in. In their book, Skelton and Pais discuss three interaction modes, which describe different levels of coordination:

Collaboration
: This interaction mode requires both teams to work closely together. Collaboration provides opportunities for teams to learn, discover, and innovate. But it requires high levels of coordination from each team and is difficult to scale. For example, a security team might collaborate with a microservices team to develop a more secure version of their software. The collaborative work might entail designing, writing, and testing code together.

Facilitating
: A facilitating interaction is similar to a collaborative one, but it is unidirectional. Instead of teams working together to solve a shared problem, one team plays a support role to help the other team deliver their desired outcome. An example of a facilitating interaction would be when an infrastructure team helps a microservices team understand how to troubleshoot issues with the network architecture they've been provided.

X-as-a-service
: Sometimes team collaboration takes on a consumer-provider flavor. In this type of interaction, one team provides a service to other teams in the organization with minimal levels of coordination. This usually occurs when a team releases a shared process, document, library, API, or platform. *X*-as-a-service interactions tend to scale well because they require less coordination. They are also a natural fit for platform teams, but other team types may incorporate this mode as well. For example, an enabling architecture team might document a list of recommended software patterns and offer those to all microservices teams in a "patterns as a service" model.

There's a lot more to Team Topologies then we've outlined here. Taken together, this categorization of team types and interactions gives us a great palette of terms we can use to paint a picture of what our microservices teams should look like, with particular emphasis on when and how much our teams will need to coordinate. In the next section, we'll use the terms we've borrowed from Team Topology to design a microservices team model.

Designing a Microservices Team Topology

The Team Topology approach gives us a language for talking about team coordination. What makes it really special, is that it's a language built for visual representations. In this section, we're going to create a design for our microservices teams that communicates the teams we need and how they will work together. When we're done, we'll have a diagram that highlights the main points of team coordination and interaction.

To create a team design and Team Topology, we'll follow this step-by-step approach:

1. Establish a system design team.
2. Create a microservices team template for future teams.
3. Define platform teams.
4. Add enabling and complicated-subsystem teams.
5. Add key consumer teams.

As we go through each of the steps, we'll be documenting our team design and building our Team Topology. For each step we'll identify one or more teams, create and populate a team design document, and draw the key interactions for that team. Let's get started by focusing on the system design team.

There isn't a single Team Topology that is a good fit for everyone. It would be impossible to account for your organization's size, people, skills, and needs. The topology we've created here is a consolidated version of large enterprise-scale implementations that we've seen work well.

Establish a System Design Team

A microservices system is a complex system with lots of parts and lots of people doing work. The software that gets built emerges from the collective decision making and work of all those people together. In our experience, getting everything to work together the way you want isn't easy. That's why you'll need to designate a group of people who can shape the vision and behavior of the system. In our model, we'll call this group the system design team.

In our model, the system design team has three core reponsibilities:

Design team structures
: The system design team is the first team we're putting together. It's also the team that we expect to design the teams that will do the work of building the system. That's the work we'll be doing in our subsequent team design steps. In effect, we're playing the role of the system design team together.

Establish standards, incentives, and "guardrails"
: In addition to forming teams, the system design team should shape the decisions that individual teams can make. This ensures that teams produce results that align with our system goals. One way to do this is by enacting standards that dictate what teams can and can't do. That's the prescriptive approach we've taken for many of the decisions in this book. In practice, too much standardization is difficult to maintain and too restrictive for a healthy system. Good designers will introduce incentives to get more of the behavior they want and "guardrails" that act as lighter recommendations and references rather than outright rules.

Continually improve the system
: Finally, the system design team needs to continually improve all the team designs, standards, incentives, and guardrails that have been introduced. To do that, they'll need to establish a way of monitoring or measuring the system as a whole so that they can make changes and introduce improvements.

It's useful to document these team responsibilities so that we can clearly communicate what each team does. In fact, we should document all of the key properties of our teams to make it easier to understand and improve them as the system evolves. At a minimum, we should cover the Team Topology type, the size of the team, and the responsibilities we've defined earlier.

Let's start by deciding on a Team Topology type. After the initial setup of team designs and standards, we expect the system design team to focus on helping other teams build microservices and supporting components. We expect most of their work to be consulting based, facilitating delivery teams and helping them navigate the system. Although the system design team delivers a system, the work we want them to do is characteristic of an enabling team type.

We also want the system design team to be small. It should consist of just a few senior leaders, architects, and system designers who can quickly make decisions together for the system as a whole. To that end, we'll limit the size of the team to between three to five people—even less than our general team size that we decided on earlier.

Let's capture these decisions and team properties by creating a lightweight design document for the system design team. Using your favorite text or document editor, create a file named *system-design-team.md* and populate it with the following content:

```
# System Design Team

## Team Type
Enabling

## Team Size
3-5 People

## Responsibilities
* Design team structures
* Establish standards and "guardrails"
* Continually improve the system
```

The nice thing about using a text file for our team documentation is that we can treat it like code. Because of this, we can store the documentation in a code repository and version it whenever we need to make changes. Alternatively, you can use a wiki, document repository, or whatever works best in your company. We'll leave it to you to decide how you want to manage your team design files. You can find all of the examples for our team designs in our GitHub repository.

At this point, we'd typically diagram the team visually and map out its interactions with other teams in the system. This is the heart of our team design work and allows us to visualize how teams will work together. For example, we can expect the system design team to use a facilitating interaction model with microservices teams. But since this is the first team we've defined, we don't have anything to interact with. So we'll leave the diagramming work for later.

With the system design team document created, we can move on to documenting and diagramming our microservices teams.

Building a Microservices Team Template

In the "up and running" model, every microservice is owned by a team. This single team owns the decisions and work of designing, building, delivering, and maintaining a microservice. In practice, a single team may own multiple microservices. This is fine, and avoids unnecessary growth of teams. The most important constraint is that the responsibility for a microservice is not shared across multiple teams. Microservice ownership will be limited to an accountable and responsible team.

Key Decision: Microservice Ownership

Each microservice will be owned by a single team, who will design, build, and run it. This team is responsible for the microservice for the lifetime of the service.

As your system matures, you'll end up with lots of microservices. You'll also likely end up with lots of microservices teams. Since we expect to have multiple microservices teams operating in our system, we won't design each of them individually. Instead, we'll define a microservices team template that can be applied to any new teams we create. Think of this as creating a cookie cutter that we can use to "punch out" some microservices teams later on when we need them. Or, if you have a programming background, you can think of this as defining a "class," for which we'll be creating "instances" later.

To get started, we'll do the same thing we did for our system design team—define some essential team properties. Just like before, we'll document the team type, team size, and responsibilities. As we mentioned before, our microservices teams are expected to own one or more microservices independently. That ownership includes running the service and releasing a continuous stream of improvements, fixes, and changes as needed.

With that characteristic, it makes sense to classify the microservices team as stream-aligned. We'll also stick to the team-sizing decision we made earlier in this chapter and keep the team size between five to eight people. Let's document all of these properties like we did before. Create a file named *microservice-team-template.md* and populate it with the following content:

```
# Microservices Team Template

## Team Type
Stream-Aligned

## Team Size
5-8 People

## Responsibilities
* Designing and developing microservice(s)
* Testing, building, and delivering the microservice(s)
* Troubleshooting issues
```

With the template definition documented, we can start diagramming our team interaction model. To do that, open a drawing or diagramming tool and draw a horizontal rectangle as shown in Figure 2-1. We have used yellow for this; each type of team should have its own color.

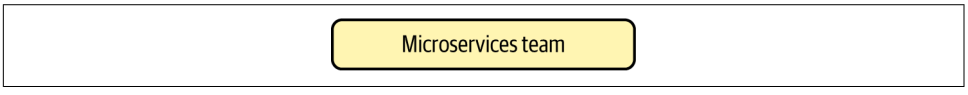


Figure 2-1. A stream-aligned microservices team

If you don't have a favorite diagramming tool, diagrams.net and Lucidchart are good browser-based options that are free to get started with. Of course, you're also free to diagram the old-fashioned way, with a pen and a napkin!

In the previous section we defined our system design team. Now that we have our microservices team diagrammed, we can add the systems team into the picture. Draw the system design team using a vertical rectangle, as shown in Figure 2-2.

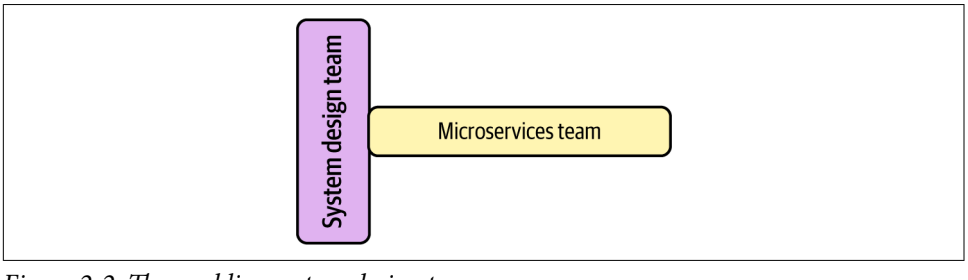


Figure 2-2. The enabling system design team

Use a unique color (we have used violet) for the system design team to denote that it's an enabling team. We've placed it vertically and to the left of the microservices team to show an interaction between the two teams. In this case, we expect the system design team to facilitate the microservices teams. To keep things simple, we aren't going to model the specific details of the interaction mode. Highlighting that the teams will need to interact is enough now.

Our color choices for the team types in this chapter are based on the illustrations shown on the Team Topologies website.

In practice, as your system evolves, you'll need to replace this generic "Microservices team" box with the actual names of your teams and the services they are working on.

Over time, you may also need to capture the interactions that must take place between your microservices teams. For example, if one microservice needs to invoke another service, chances are there will be some coordination work that is worth capturing.

We use a particular color to denote that our microservices team is stream-aligned. We'll be updating this diagram as we go through the team design steps, so keep your drawing tool handy for later on. You may also want to save the diagram so you don't lose any work.

Now that we have our first two teams modeled, let's take a look at the cloud platform team.

Platform Teams

Platform teams are an important part of a microservices system. Most of the microservices work is done by independent, stream-aligned teams. Without support, however, they'll need to figure out how to solve a lot of development, testing, and implementation problems on their own. Our facilitating system design team can enable some of their decision making, but the microservices teams will still need to deal with the complexities of an entire technology stack and architecture.

That's where platform team types can help. There are a lot of common components in a microservices system. A platform team can make those common components available for microservices to use "as a service." The service model improves the scalability of platform components, reducing the coordination problems that usually occur when shared components are centralized.

In our model, we've decided to instantiate a cloud platform team that offers a network, application, and deployment infrastructure to the rest of the organization as a service. We'll get into the details of what this offering looks like in Chapter 7, when we dive into infrastructure design. The key point for now is that the teams in our system will be able to create new environments on demand using the infrastructure services that our platform team provides.

With those details understood, we can document our cloud platform team in a file called *cloud-platform-team.md* with the following team properties:

```
# Cloud Platform Team

## Team Type
Platform

## Team Size
5-8 People

## Responsibilities
* Design and develop a network infrastructure
```

```
* Design and develop an application infrastructure
* Provide tools for building a new environment
* Update network and application infrastructure when required
```

Notice that one of the responsibilities of our cloud platform team is to update the infrastructure that is being offered. This is a key part of a platform team's responsibility. They need to treat the users of the platform as if they are customers. In this relationship, the platform offering needs to be continually improved to meet their customer's requirements and expectations.

As we've done before, we'll add the cloud platform team to the Team Topology diagram that we've been working on. But this time we need to model a platform team. To do that, draw a horizontal rectangle (again, using a unique color; we've used light blue) below the microservices teams and connected to the system design team, as shown in Figure 2-3.

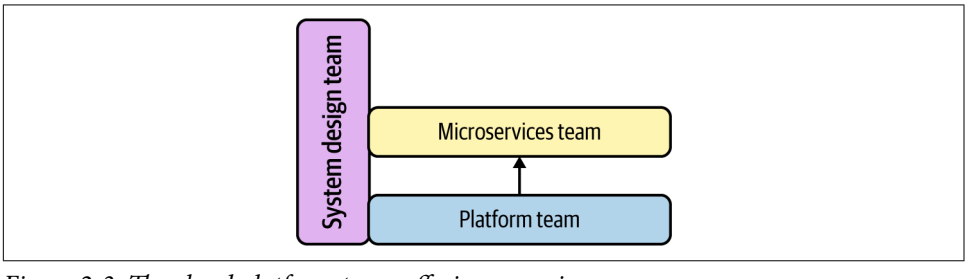


Figure 2-3. The cloud platform team offering a service

Note that we've also drawn a small black arrow between the platform and microservices teams. This is to show that the platform team is implementing the *x*-as-a-service model for its interaction with the microservices team. Our diagram also shows that the system design team will be enabling the work of the platform team. This will ensure that the platform fits the goals and vision for the overall system.

For our "up and running" model, we've only defined a single instance of a platform team. But, in practice, you'll probably need to roll out multiple platform teams to keep the teams to a manageable size. When that happens, you'll also need to be consider how multiple platform teams will coordinate together to offer services to the rest of the organization.

Enabling and Complicated-Subsystem Teams

With the three teams we've designed, we have enough people in place to be able to deliver a microservices system. Beyond these core capabilities, there may be additional capabilities that we want a team to own. That may be because there is an important set of skills that we want to provide enablement for. Or, because there is a complicated system feature that requires a dedicated team.

In our microservices model, we've decided to create a specialized release team. This additional team owns the responsibility of releasing (or deploying) microservices into a production-like environment. While a microservices team could deploy its own services directly into a production environment, in our experience this isn't always what happens.

That's because in most organizations there is usually an additional testing and acceptance check that needs to happen before a service can go live. Instead of deploying directly into production, microservices teams deliver a built and tested container. That container is then automatically deployed by a release team who coordinates the work of tests, approvals, and deployment of the change.

The release team embodies the complicated-subsystem team type. It contains specialist knowledge of the release, approval, and deployment process and collaborates with stream-aligned teams to make that work happen. To document the design of our release team, create a file called *release-team.md* with the following properties:

```
# Release Team

## Team Type
Complicated-Subsystem

## Team Size
5-8 People

## Responsibilities
* Releasing microservices to production
* Coordinating approvals for releases
```

Next, we'll add the release team to the developing picture of our Team Topology. Complicated-subsystem teams are modeled with yet another specific color.

So, open your Team Topology diagram and add a square (we've colored it red) near the end of the microservices team's box, as shown in Figure 2-4.

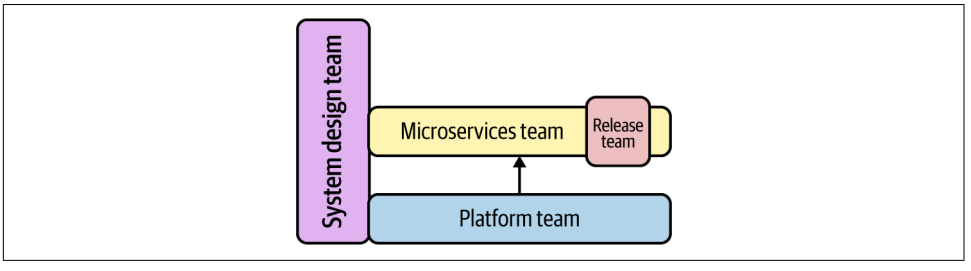


Figure 2-4. The release team

As we can see from our emerging topology, one of the trade-offs to the release team approach is the coordination costs it brings. At scale, this can become a big problem. For example, if you want to perform daily releases across multiple microservices, the release team will struggle to coordinate all of that activity. If you find yourself in that situation, you'll need to change the team design and shift the responsibilities for deployment to the individual microservices teams.

The release team is the final team at the core of our microservices model. But to finish our design, we need to consider the teams that will have to use our microservices. We'll cover that next.

Consumer Teams

Microservices are only useful if they are used. So, it's worthwhile identifying the consumers of our microservices and how they'll interact with our system teams. In some architectures, that could include mobile application development teams, web development teams, or even third-party organizations. In our model, the main consumer of our microservices system is the API team.

The API team is responsible for exposing our microservices to other development teams as an application programming interface (API). For example, a mobile application development team would interact with the API released by this team and never call our microservices directly.

We'll get into the details of the API and the architecture later in the book. For now, it's worth detailing the properties of our API team and its responsibilities. We can do that by creating a file named *api-team.md* and populating it as follows:

```
# API Team

## Team Type
Stream-Aligned

## Team Size
5-8 People

## Responsibilities
* Design, develop, and maintain APIs at the boundary of the system
* Connect API to internal microservices
```

Just like the microservices team, the API team is a stream-aligned team. That's because it needs to continually deliver changes to the API that reflect business needs and consumer demands. A special nuance of the API team is that, because the API needs to call microservices to function, it is dependent on the microservices team.

We can model these interaction properties in our Team Topology model by adding another rectangle at the top of our diagram to represent the API team. It should be of the same color as the microservices team (we've used yellow), as it is also a stream-aligned team. To reflect the dependency between our microservices and API teams, we'll again use a black arrow to show an *x*-as-a-service engagement model. This indicates that the microservices team will need to make sure their services are invocable and usable in a self-service fashion.

When you're finished, the diagram should look something like Figure 2-5.

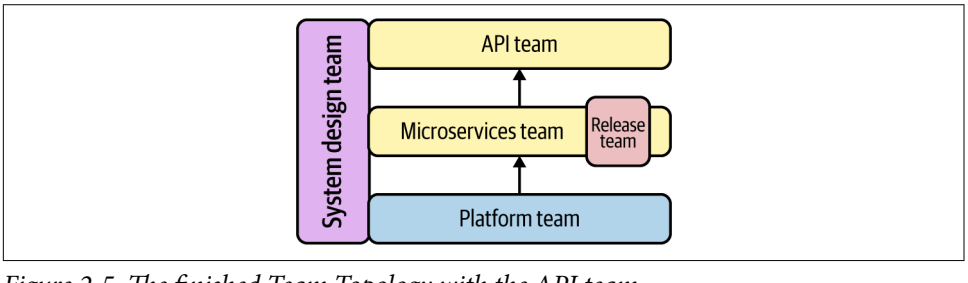


Figure 2-5. The finished Team Topology with the API team

With this final team defined an in the picture, our topology looks a lot like the finished product we showed you at the start of this section. With the topology defined, we can see where the main coordination points are where the work is being done. Overall, our model enables a fairly independent, autonomous way of working. However, we'll need to invest some time and effort into building a cloud platform as a service to make our model work.

With our basic Team Topology defined, we can see how this work ties back to the goal of our chapter—building an operating model.

Summary

Taken together, the decisions, team definitions, and topologies we've just created form our microservices operating model. With it, we've defined the teams that need to be created, their characteristics and responsibilities, and the way we expect our teams to work together. It's an important design step and will influence the rest of our microservices work. In fact, every decision we make from this point on will be heavily influenced by the operating model we've just established.

In truth, we didn't go very deep with our operating model design. In practice, it's worthwhile drawing out more than one Team Topology diagram to reflect different types of interaction modes. For example, the toubleshooting problems with our system would likely require a different engagement model from the one we've shown. Similarly, we haven't diagrammed the interactions required to change the deliverables that the cloud platform, system design, and release teams provide.

In addition to the initial design, the operating model design should be continually improved. One of the nice things about capturing our team definitions and topologies as documents is that we can treat them like code. So we can version and manage changes as the system evolves. You may even want to add additional design assets to your collection. For example, you could define service-level agreements for platform teams and skill inventories for your stream-aligned teams.

Ultimately, our goal in this chapter was to create a foundation for the rest of our design and development work. Our lightweight approach to the Team Topology and team designs does just that. With our operating model in hand, we can move on to designing the actual microservices. That's what we'll cover in Chapter 3.

CHAPTER 4

Rightsizing Your Microservices: Finding Service Boundaries

One of the most challenging aspects of building a successful microservices system is the identification of proper microservice boundaries. It makes intuitive sense that breaking up a large codebase into smaller, simpler, more loosely coupled parts improves maintainability, but how do we decide where and how to split the code into parts to achieve those desired properties? What rules do we use to know where one service ends and another one starts? Answering these fundamental questions is challenging. A lot of teams new to microservices stumble at them. Drawing the microservice boundaries incorrectly can significantly diminish the benefits of using microservices, or in some cases even derail the entire effort. It is then not surprising that the most frequent, most pressing question microservices practitioners ask is: how can a bigger application be properly sliced into a collection of microservices?

In this chapter, we look deep into the leading methodology for the effective analysis, modeling, and decomposition of large domains (Domain-Driven Design), explain the efficiency benefits of using Event Storming for domain analysis, and close by introducing the Universal Sizing Formula, a unique guidance for the effective sizing of microservices.

Why Boundaries Matter, When They Matter, and How to Find Them

Right in the title of the architectural pattern, we have the word *micro*—the architecture we are designing is that of "micro" services! But how "micro" should our services be? We are obviously not measuring the physical length of something and assuming that *micro* means one-millionth of a meter (i.e., of the base unit of length in the International System of Units). So what does *micro* mean for our purposes? How are we

supposed to slice up our larger problem into smaller services to achieve the promised benefits of "micro" services? Maybe we could print our source code on paper, glue everything together, and measure the literal length of that? Or jokes aside, should we go by the number of lines in our source code—keeping that number small to ensure each of our microservices is also small enough? What is "enough," however? Maybe we just arbitrarily declare that each microservice must have no greater than 500 lines of code? We could also draw boundaries at the familiar, functional edges of our source code and say that each granular capability represented by a function in the source code of our system is a microservice. This way we could build our entire application with, say, serverless functions, declaring each such function to be a microservice. Clean and easy! Right? Maybe not.

In practice, each of these simplistic approaches has indeed been tried and they all have significant drawbacks. While source lines of code (SLOC) has historically enjoyed some usage as a measure of effort/complexity, it has since been widely acknowledged to be a poor measurement for determining the complexity or the true size of any code and one that can be easily manipulated. Therefore, even if our goal were to create "small" services with the hope of keeping them simple, lines of code would be a poor measurement.

Drawing boundaries at functional edges is even more tempting. And it has become even more tempting with the increase in popularity of serverless functions such as Amazon Web Services' Lambda functions. Building on top of the productivity and wide adoption of AWS Lambdas, many teams have rushed into declaring those functions "microservices." There are a number of significant problems if you go down this road, the most important of which are:

Drawing boundaries based on technical needs is an anti-pattern
: Per Lewis and Fowler, microservices should be "organized around business capabilities," not technical needs. Similarly, Parnas, in an article from 1972, recommends decomposing systems based on modular encapsulation of design changes over time. Neither approach necessarily aligns strongly with the boundaries of serverless functions.

Too much granularity, too soon
: An explosive level of granularity early in the microservices project life cycle can introduce crushing levels of complexity that will stop the microservices effort in its tracks, even before it has a chance to take off and succeed.

In Chapter 1 we stated the primary goal of a microservices architecture: it is primarily about minimization of coordination costs, in a complex, multiteam environment, to achieve harmony between speed and safety, at scale. Therefore, services should be designed in a way that minimizes coordination needs between the teams working on different microservices. However, if we break code up into functions in a way that does not necessarily lead to minimized coordination, we will end up with incorrectly

sized microservices. Just assuming that any way of organizing code into serverless functions will reduce coordination is misguided.

Earlier we stated that an important reason for avoiding a size-based or functions-aligned approach when splitting an application into microservices is the danger of premature optimization—having too many services that are too small too early in your microservices journey. Early adopters of microservices, such as Netflix, SoundCloud, Amazon, and others, eventually found themselves having a lot of microservices! That, however, does not mean that these companies started with hundreds of very granular microservices on day one. Rather, a large number of microservices is what they optimized for after years of development, *after* having achieved the operational maturity capable of handling the level of complexity associated with the high granularity of microservices.

Avoid Creating Too Many Microservices Too Early

The sizing of services in a microservices architecture is most certainly a journey that should unfold in time. A sure way to sabotage the entire effort is to attempt designing an overly granular system early in that journey.

Whether you are working on a greenfield project or decomposing an existing monolith, the approach should absolutely be to start with only a handful of services and slowly increase the number of microservices over time. If this leads to some of your microservices initially being larger than in their target state, it is totally OK. You can split them up later.

Even if we are starting with just a few microservices, taking it slow, we need some reliable methodology to determine how to size microservices. Next, we will explore best practices successfully used in the industry.

Domain-Driven Design and Microservice Boundaries

At the onset of figuring out microservices design best practices, Sam Newman introduced some foundational ground rules in his book *Building Microservices* (O'Reilly). He suggested that when drawing service boundaries, we should strive for such a design that the resulting services are:

Loosely coupled
: Services should be fairly unaware and independent of each other, so that a code modification in one of them doesn't result in ripple effects in others. We'll also probably want to limit the number of different types of runtime calls from one service to another since, beyond the potential performance problem, chatty communications can also lead to a tight coupling of components. Taking our "coordination minimization" approach, the benefit of the loose coupling of the services is quite obvious.

Highly cohesive
: Features present in a service should be highly related, while unrelated features should be encapsulated elsewhere. This way, if you need to change a logical unit of functionality, you should be able to change it in one place, minimizing time to releasing that change (an important metric). In contrast, if we had to change the code in a number of services, we would have to release lots of different services at the same time to deliver that change. That would require significant levels of coordination, especially if those services are "owned" by multiple teams, and it would directly compromise our goal of minimizing coordination costs.

Aligned with business capabilities
: Since most requests for the modification or extension of functionality are driven by business needs, if our boundaries are closely aligned with the boundaries of business capabilities, it would naturally follow that the first and second design requirements, above, are more easily satisfied. During the days of monolith architectures, software engineers often tried to standardize on "canonical data models." However, the practice demonstrated, over and over again, that detailed data models for modeling reality do not last for long—they change quite often and standardizing on them leads to frequent rework. Instead, what is more durable is a set of business capabilities that your subsystems provide. An accounting module will always be able to provide the desired set of capabilities to your larger system, regardless of how its inner workings may evolve over time.

These design principles have proven to be very useful and received wide adoption among microservices practitioners. However, they are fairly high-level, aspirational principles and arguably do not provide the specific service-sizing guidance needed by day-to-day practitioners. In search of a more practical methodology, many turned to Domain-Driven Design.

The software design methodology known as *Domain-Driven Design* (DDD) significantly predates microservices architecture. It was introduced by Eric Evans in 2003, in his seminal book of the same name, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley). The main premise of the methodology is the assertion that, when analyzing complex systems, we should avoid seeking a single

unified domain model representing the entire system. Rather, as Evans said in his book:

> Multiple models coexist on big projects, and this works fine in many cases. Different models apply in different contexts.

Once Evans established that a complex system is fundamentally a collection of multiple domain models, he made the critical additional step of introducing the notion of *bounded context*. Specifically, he stated that:

> A Bounded Context defines the range of applicability of each model.

Bounded contexts allow implementation and runtime execution of different parts of the larger system to occur without corrupting the independent domain models present in that system. After defining bounded contexts, Eric went on to also helpfully provide a formula for identifying the optimal edges of a bounded context by establishing the concept of *Ubiquitous Language*.

To understand the meaning of Ubiquitous Language, it is important to observe that a well-defined domain model first and foremost provides a common vocabulary of defined terms and notions, a common language for describing the domain, that subject-matter experts and engineers develop together in close collaboration, balancing the business requirements and implementation considerations. This common language, or shared vocabulary, is what in DDD we call Ubiquitous Language. The importance of this observation lies in acknowledging that same words may carry different meanings in different bounded contexts. A classic example of this is shown in Figure 4-1. The term *account* carries significantly different meaning in the identity and access management, customer management, and financial accounting contexts of an online reservation system.

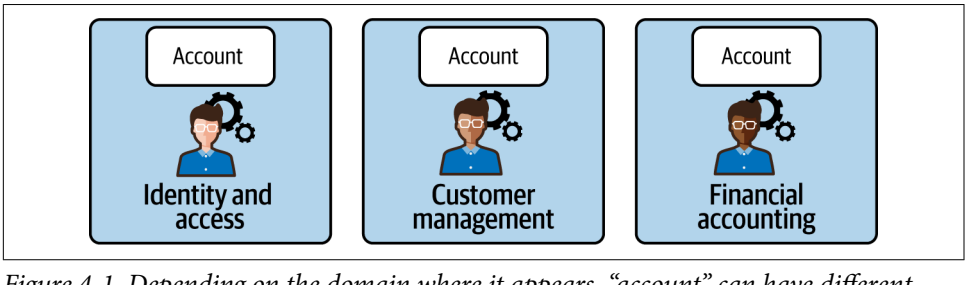


Figure 4-1. Depending on the domain where it appears, "account" can have different meanings

Indeed, for an identity and access management context, an account is a set of credentials used for authentication and authorization. For a customer management-bounded context, an account is a set of demographic and contact attributes, while for a financial accounting context, it's probably payment information and a list of past transactions. We can see that the same basic English word is used with significantly

different meaning in different contexts, and it is OK because we only need to agree on the ubiquitous meaning of the terms (the Ubiquitous Language) within the bounded context of a specific domain model. According to DDD, by observing edges across which terms change their meaning, we can identify the boundaries of the contexts.

In DDD, not all terms that come to mind when discussing a domain model make into the corresponding Ubiquitous Language. Concepts in a bounded context that are core to the context's primary purpose are part of the team's Ubiquitous Language, all others should be left out. These core concepts can be discovered from the set of JTBDs that you create for the bounded context. As an example, let's look at Figure 4-2.

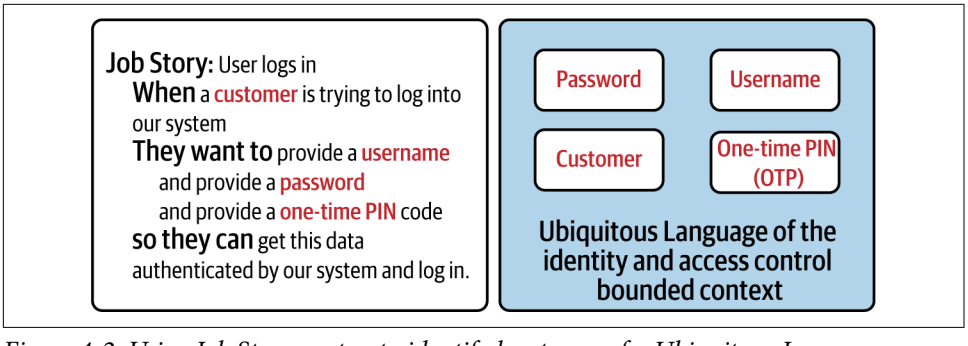


Figure 4-2. Using Job Story syntax to identify key terms of a Ubiquitous Language

In this example, we are using the Job Story format that we introduced in Chapter 3 and applying it to a job from the identity and access control bounded context. We can see that key nouns, highlighted in Figure 4-2, correspond to the terms in the related Ubiquitous Language. We highly recommend the technique of using key nouns from well-written Job Stories in the identification of the vocabulary terms relevant to your Ubiquitous Language.

Now that we have discussed some key concepts of DDD, let's also look at something that can be very useful in designing microservice interactions properly: context mapping. We will explore key aspects of context mapping in the next section.

Context Mapping

In DDD, we do not attempt to describe a complex system with a single domain model. Rather, we design multiple independent models that coexist in the system. These subdomains typically communicate with each other using published interface descriptions. The representation of various domains in a larger system and the way they collaborate with each other is called a *context map*. Consequently, the act of identifying and describing said collaborations is known as *context mapping*, as shown in Figure 4-3.

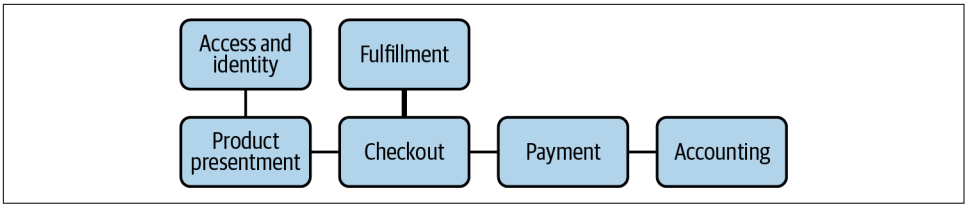


Figure 4-3. Context mapping

DDD identifies several major types of collaboration interactions when mapping bounded contexts. The most basic type is known as a *shared kernel*. It occurs when two domains are developed largely independently and, almost by accident, they end up overlapping on some subset of each other's domains (see Figure 4-4). Two parties may agree to collaborate on this shared kernel, which may also include shared code and data model, as well as the domain description.

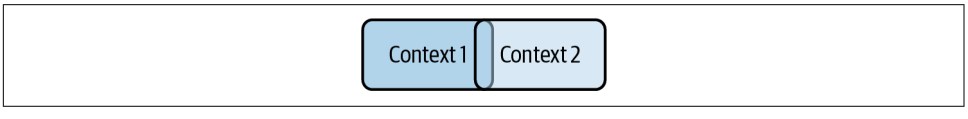


Figure 4-4. Shared kernel

While tempting on the surface of things (after all, the desire for collaboration is one of the most human of instincts), the shared kernel is a problematic pattern, especially when used for microservices architectures. By definition, a shared kernel immediately requires a high degree of coordination between two independent teams to even jump-start the relationship, and keeps requiring coordination for any further modifications. Sprinkling your microservices architecture with shared kernels will introduce many points of tight coordination. In cases when you do have to use a shared kernel in a microservices ecosystem, it's advised that one team is designated as the primary owner/curator, and everybody else is a contributor.

Alternatively, two bounded contexts can engage in what DDD calls an Upstream–Downstream kind of relationship. In this type of relationship, the Upstream acts as the provider of some capability, and the Downstream is the consumer of said capability. Since domain definitions and implementations do not overlap, this type of relationship is more loosely coupled than a shared kernel (see Figure 4-5).

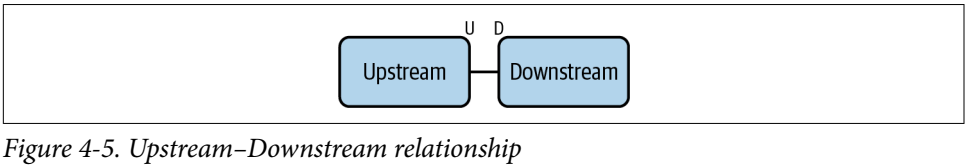


Figure 4-5. Upstream–Downstream relationship

Depending on the type of coordination and coupling, an Upstream–Downstream mapping can be introduced in several forms:

Customer–Supplier
: In a customer–supplier scenario, Upstream (supplier) provides functionality to the Downstream (customer). As long as the provided functionality is valuable, everybody is happy; however, Upstream carries the overhead of backwards compatibility. When Upstream modifies their service, they need to ensure that they do not break anything for the customer. More dramatically, the Downstream (customer) carries the risk of the Upstream intentionally or unintentionally breaking something for it, or ignoring the customer's future needs.

Conformist
: An extreme case of the risks for a customer–supplier relationship is the *conformist* relationship. It's a variation on Upstream–Downstream, when the Upstream explicitly does not or cannot care about the needs of its Downstream. It's a use-at-your-own-risk kind of relationship. The Upstream provides some valuable capability that the Downstream is interested in using, but given that the Upstream will not cater to its needs, the Downstream needs to constantly conform to the changes in the Upstream.

Conformist relationships often occur in large organizations and systems when a much larger subsystem is used by a smaller one. Imagine developing a small, new capability inside an airline reservation system and needing to use, say, an enterprise payments system. Such a large enterprise system is unlikely to give the time of day to some small, new initiative, but you also cannot just reimplement a whole payments system on your own. Either you will have to become a conformist, or another viable solution may be to *separate ways*. The latter doesn't always mean that you will implement similar functionality yourself. Something like a payments system is complex enough that no small team should implement it as a side job of another goal, but you might be able go outside the confines of your enterprise and use a commercially available payments vendor instead, if your company allows it.

In addition to becoming a conformist or going separate ways, the Downstream has a few more DDD-sanctioned ways of protecting itself from the negligence of its Upstream: an anti-corruption layer and using Upstreams that provide open host interfaces.

Anti-corruption layer
: In this scenario, the Downstream creates a translation layer called an *anti-corruption layer* (ACL) between its and the Upstream's Ubiquitous Languages, to guard itself from future breaking changes in the Upstream's interface. Creating an ACL is an effective, sometimes necessary, measure of protection, but teams should keep in mind that in the long term this can be quite expensive for the Downstream to maintain (see Figure 4-6).

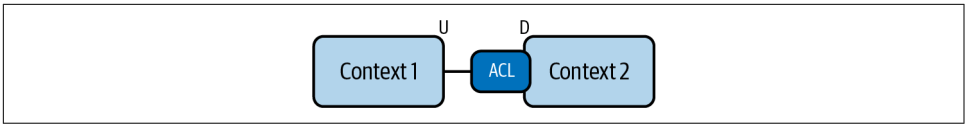


Figure 4-6. Anti-corruption layer

Open host service
: When the Upstream knows that multiple Downstreams may be using its capabilities, instead of trying to coordinate the needs of its many current and future consumers, it should instead define and publish a standard interface, which all consumers will need to adopt. in DDD, such Upstreams are known as open host services. By providing an open, easy protocol for all authorized parties to integrate with, and maintaining said protocol's backwards compatibility or providing clear and safe versioning for it, the open host can scale its operations without much drama. Practically all public services (APIs) use this approach. For example, when you are using the APIs of a public cloud provider (AWS, Google, Azure, etc.), they usually don't know or cater to you specifically as they have millions of customers, but they are able to provide and evolve a useful service by operating as an open host (see Figure 4-7).

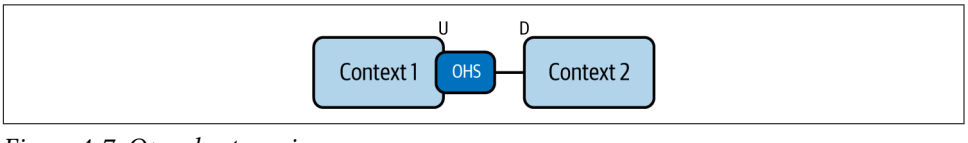


Figure 4-7. Open host service

In addition to relation types between domains, context mappings can also differentiate based on the integration types used between bounded contexts.

Synchronous Versus Asynchronous Integrations

Integration interfaces between bounded contexts can be synchronous or asynchronous, as shown in Figure 4-8. None of the integration patterns fundamentally assume one or the other style.

Common patterns for synchronous integrations between contexts are RESTful APIs deployed over HTTP, gRPC services using binary formats such as protobuf, and more recently services using GraphQL interfaces.

On the asynchronous side, publish–subscribe types of interactions lead the way. In this interaction pattern, the Upstream can generate events, and Downstream services have workers able and interested in processing those, as depicted in Figure 4-8.

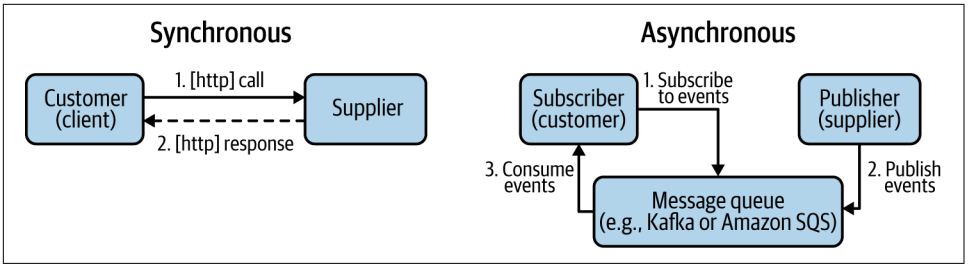


Figure 4-8. Synchronous and asynchronous integrations

Publish–subscribe interactions are more complex to implement and debug, but they can provide a superior level of scalability, resilience, and flexibility, in that: multiple receivers, even if implemented with heterogeneous tech stack, can subscribe to the same events using a uniform approach and implementation.

To wrap up the discussion of Domain-Driven Design's key concepts, we should explore the concept of an *aggregate*. We discuss it in the next section.

A DDD Aggregate

In DDD, an *aggregate* is a collection of related domain objects that can be viewed as a single unit by external consumers. Those external consumers only reference a single entity in the aggregate, and that entity is known in DDD as an *aggregate root*. Aggregates allow domains to hide internal complexities of a domain, and expose only information and capabilities (interface) that are "interesting" to an external consumer. For instance, in the Upstream–Downstream mappings that we discussed earlier, the Downstream does not have to, and typically will not want to, know about every single domain object within the Upstream. Instead, it will view the Upstream as an aggregate, or a collection of aggregates.

We will see the notion of an aggregate resurface, in the next section when we discuss Event Storming—a powerful methodology that can greatly streamline the process of domain-driven analysis and turn it into a much faster and more fun exercise.

Introduction to Event Storming

Domain-Driven Design is a powerful methodology for analyzing both the whole-system-level (called "strategic" in DDD) as well as the in-depth (called "tactical") composition of your large, complex systems. We have also seen that DDD analysis can help us identify fairly autonomous subcomponents, loosely coupled across bounded contexts of their respective domains.

It's very easy to jump to the conclusion that in order to fully learn how to properly size microservices, we just need to become really good in domain-driven analysis; if

we make our entire company also learn and fall in love with it (because DDD is certainly a team sport), we'll be on our way to success!

In the early days of microservices architectures, DDD was so universally proclaimed as *the one true way* to size microservices that the rise of microservices gave a huge boost to the practice of DDD, as well—or at least more people became aware of it, and referenced it. Suddenly, many speakers were talking about DDD at all kinds of software conferences, and a lot of teams started claiming that they were employing it in their daily work. Alas, a close look easily uncovered that the reality was somewhat different and that DDD had become one of those "much-talked-about-less-practiced" things.

Don't get us wrong: there were people using DDD way before microservices, and there are plenty using it now as well, but speaking specifically of using it as a tool for sizing microservices, it was more hype and vaporware than reality.

There are two primary reasons why more people talked about DDD than practiced it in earnest: it is complex and it is expensive. Practicing DDD requires quite a lot of knowledge and experience. Eric Evans's original book on the subject is a hefty 520 pages long, and you would need to read at least a few more books to really get it, not to mention gain some experience actually implementing it on a number of projects. There simply were not enough people with the skills and experience and the learning curve was steep.

To exacerbate the problem, as we mentioned, DDD is a team sport, and a time-consuming one at that. It's not enough to have a handful of technologists well-versed in DDD; you also need to sell your business, product, design, etc., teams on participating in long and intense domain-design sessions, not to mention explain to them at least the basics of what you are trying to achieve. Now, in the grand scheme of things, is it worth it? Very likely, yes: especially for large, risky, expensive systems, DDD can have many benefits. However, if you are just looking to move quickly and size some microservices, and you have already cashed in your political capital at work, selling everybody on the new thing called microservices—good luck also asking a whole bunch of busy people to give you enough time to size your services right! It was just not happening—too expensive and too time-consuming.

And then suddenly a fellow by the name of Alberto Brandolini, who had invested decades in understanding better ways for teams to collaborate, found a shortcut! He proposed a fun, lightweight, and inexpensive process called Event Storming, which is heavily based and inspired by the concepts of DDD but can help you find bounded contexts in a matter of hours instead of weeks or months. The introduction of Event Storming was a breakthrough for inexpensive applicability of DDD specifically for the sake of service sizing. Of course, it's not a full replacement, and it won't give you *all* the benefits of formal DDD (otherwise it would be magic). But as far as the discovery of bounded contexts goes, with good approximation—it is indeed magical!

Event Storming is a highly efficient exercise that helps identify bounded contexts of a domain in a streamlined, fun, and efficient manner, typically much faster than with more traditional, full DDD. It is a pragmatic approach that lowers the cost of DDD analysis enough to make it viable in situations in which DDD would not be affordable otherwise. Let's see how this "magic" of Event Storming is actually executed.

Key Decision: Use Event Storming Instead of Formal DDD

Use the more lightweight Event Storming process instead of formal DDD to discover the main aggregates in your subdomain and identify edges of the various bounded contexts present in your system.

The Event-Storming Process

The beauty of Event Storming is in its ingenious simplicity. In physical spaces (preferred, when possible), all you need to hold a session of Event Storming is a very long wall (the longer the better), a bunch of supplies, mostly stickies and Sharpies, and four to five hours of time from well-represented members of your team. For a successful Event Storming session, it is critical that participants are not only engineers. Broad participation from such groups as product, design, and business stakeholders makes a significant difference. You can also host virtual Event Storming sessions using digital collaboration tools that can mimic the physical process described here.

The process of hosting physical Event Storming sessions starts by purchasing the supplies. To make things easier, we've created an Amazon shopping list that we use for Event Storming sessions (see Figure 4-9). It is comprised of:

- A large number of stickies of different colors, most importantly, orange and blue, and then several other colors for various object types. You need *a lot* of those. (Stores never had enough for me, so I got in the habit of buying online.)
- A roll of 1/2-inch white artist tape.
- A long roll of paper (e.g., IKEA Mala Drawing Paper) that we are going to hang on the wall using the artist tape. Go ahead and create multiple "lanes."
- At least as many Sharpies as the number of session participants. Everybody needs to have their own!
- Did we already mention a long, unobstructed wall that we can tape the roll of paper to?

Figure 4-9. Required supplies for an Event Storming session

During Event Storming sessions, broad participation, e.g., from subject-matter experts, product owners, and interaction designers, is very valuable. Event Storming sessions are short enough (just several hours rather than analysis requiring days or weeks) that, considering the value of their outcomes, the clarity they bring for all represented groups and the time they save in the long term, they are time well-invested for all participants. An Event Storming session that is limited to just software engineers is mostly useless, since it happens in a bubble and cannot lead to the cross-functional conversations necessary for desired outcomes.

Once we have the supplies, the large room with a wide-open wall with a roll of paper we have taped to it, and all the required people, we (the facilitator) ask everybody to grab a bunch of orange stickies and a personal Sharpie. Then we give them a simple assignment: to write the key events of the domain being analyzed as orange sticky notes (one event per one note), expressed in a verb in the past tense, and place the notes along a timeline on the paper taped to the wall to create a "lane" of time, as shown in Figure 4-10.

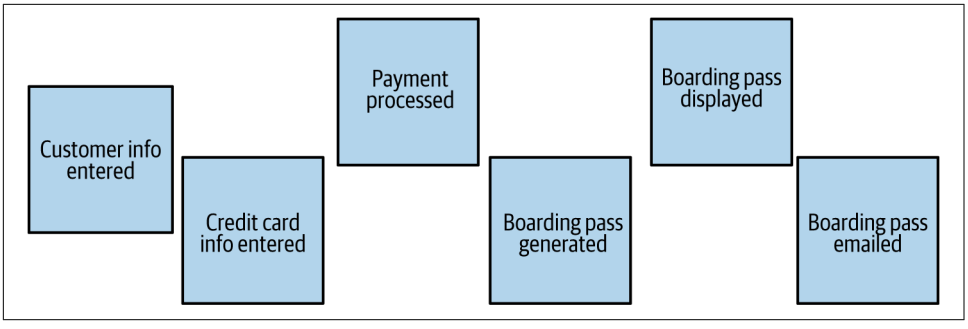


Figure 4-10. An event timeline with sticky notes

Participants should not obsess about the exact sequence of events, and at this stage there should be no coordination of events among participants. The only thing they are asked is to individually think of as many events as possible and put the events they think occur earlier in time to the left, and put the later events more to the right. It is not their job to weed out duplicates. At least, not yet. This phase of the assignment usually takes 30 minutes to an hour, depending on the size of the problem and the number of participants. Usually, you want to see at least 100 event sticky notes generated before you can call it a success.

In the second phase of the exercise, the group is asked to look at the resulting set of notes on the wall, and with the help of the facilitator, to start arranging them into a more coherent timeline, identifying and removing duplicates. Given enough time, it is very helpful for the participants to start creating a "storyline," walking through the events in an order that creates something like a "user journey." In this phase, the team may have some questions or confusion; we don't try to solve these issues, but rather capture them as "hotspots"—differently colored sticky notes (typically purple) that have the questions on them. Hotspots will need to be answered offline, in follow-ups. This phase can likewise take 30 to 60 minutes.

In the third stage, we create what in Event Storming is known as a *reverse narrative*. Basically, we walk the timeline backward, from the end to the start, and identify *commands*; things that caused the events. We use sticky notes of a different color (typically blue) for the commands. At this stage your storyboard may look something like Figure 4-11.

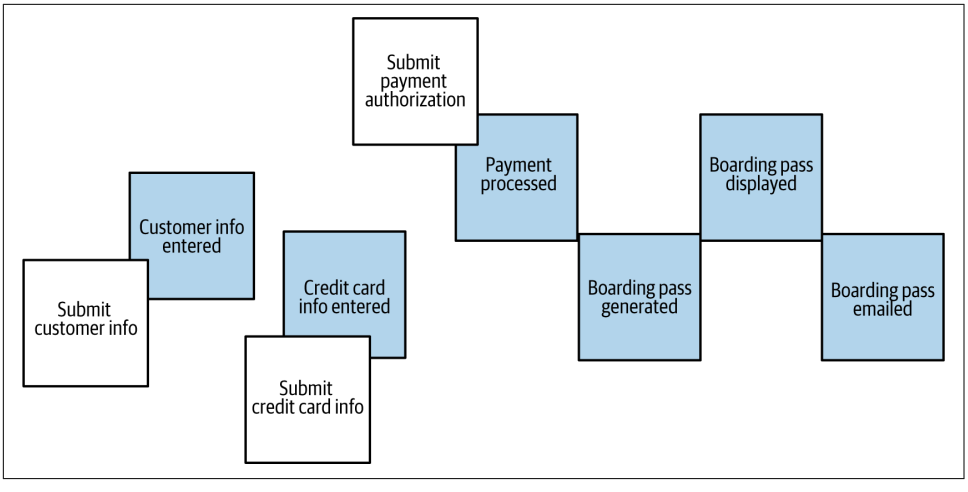


Figure 4-11. Introducing commands to the Event Storming timeline

Be aware that a lot of commands will have one-to-one relationship with an event. It will feel redundant, like the same thing worded in the past versus present. Indeed, if you look at the previous figure, the first two commands are like that. It often confuses people new to Event Storming. Just ignore it! We don't pass judgment during Event Storming, and while some commands may be 1:1 with events, some will not be. For example, the "Submit payment authorization" command triggers a whole bunch of events. Just capture what you know/think happens in real life and don't worry about making things "pretty" or "neat." The real world you are modeling is also usually messy.

In the next phase, we acknowledge that commands do not produce events directly. Rather, special types of domain entities accept commands and produce events. In Event Storming, these entities are called *aggregates* (yes, the name is inspired by the similar notion in DDD). What we do in this stage is rearrange our commands and events, breaking the timeline when needed, such that the commands that go to the same aggregate are grouped around that aggregate and the events "fired" by that aggregate are also moved to it. You can see an example of this stage of Event Storming in Figure 4-12.

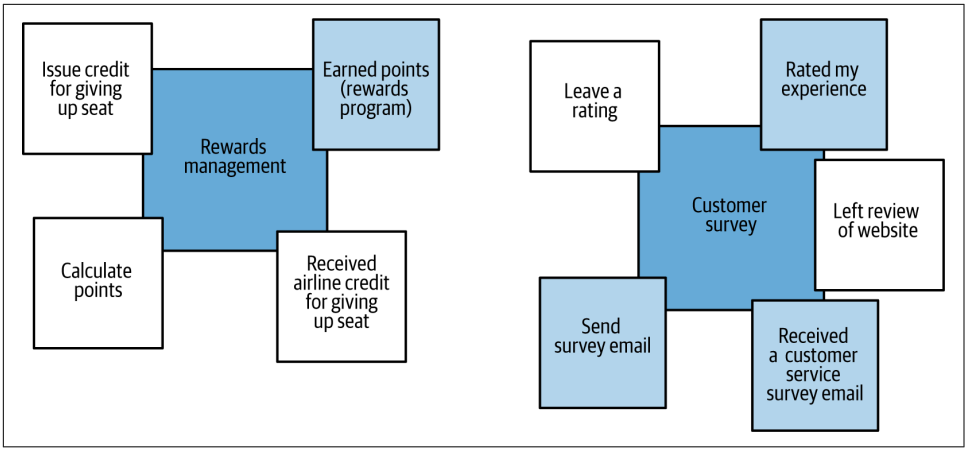


Figure 4-12. Aggregates on an Event Storming timeline

This phase of the exercise can take 15 to 25 minutes. Once we are done with it, you should discover that our wall now looks less like a timeline of events and more like a cluster of events and commands grouped around aggregates.

Guess what? These clusters are the bounded contexts we were looking for.

The only thing left is to classify various contexts by the level of their priority (similar to "root," "supportive," and "generic" in DDD). To do this, we create a matrix of bounded context/subdomains and rank them across two properties: difficulty and competitive edge. In each category, we use T-shirt sizes <S, M, or L> to rank accordingly. In the end, the decision making as to when to invest effort is based on the following guidelines:

1. Large competitive advantage/large effort: these are the contexts to design and implement in-house and spend most time on.
2. Small advantage/large effort: buy!
3. Small advantage/small effort: great assignments to trainees.
4. Other combinations are a coin toss and require a judgment call.

This last phase, the "competitive analysis," is not part of Brandolini's original Event Storming process, and was proposed by Greg Young for prioritizing domains in DDD in general. We find it to be a useful and fun exercise when done with an adequate level of humor.

The entire process is very interactive, requires the involvement of all participants, and usually ends up being fun. It will require experienced facilitator to keep things moving smoothly, but the good news is that being a good facilitator doesn't take the same effort as becoming a rocket scientist (or DDD expert). After reading this book and facilitating some mock sessions for practice, you can easily become a world-class Event Storming facilitator!

As a facilitator, it is a good idea to watch the time and have a plan for your session. For a four-hour session rough allocation of time would look like:

- Phase 1 (~30 min): Discover domain events
- Phase 2 (~45 min): Enforce the timeline
- Phase 3 (~60 min): Reverse narrative and Command Identification
- Phase 4 (~30 min): Identify aggregates/bounded contexts
- Phase 5 (~15 min): Competitive analysis

And if you noticed that these times do not add up to 4 hours, keep in mind that you will want to give people some breaks in the middle, as well as leave yourself time to prepare the space and provide guidance in the beginning.

Introducing the Universal Sizing Formula

Bounded contexts are a fantastic starting point for rightsizing microservices. We have to be cautious, however, to not assume that microservice boundaries are synonymous with the bounded contexts from DDD or Event Storming. They are not. As a matter of fact, microservice boundaries cannot be assumed to be constant over time. They evolve over time and tend to follow an increasing granularity of microservices as the organizations and applications they are part of mature. For example, Adrian Cockroft noted that this was definitely a repeating trend that they had observed during his time at Netflix.

Nobody Gets Microservice Boundaries Perfectly at the Outset

In successful cases of microservices adoption, teams do not start with hundreds of microservices. They start with a much smaller number, closely aligned with bounded contexts. As time goes by, teams split microservices when they run into coordination dependencies that they need to eliminate. This also means that teams are not expected to get service boundaries "right" out of the gate. Instead, boundaries evolve over time, with a general direction of increased granularity.

It is worth noting that it's typically easier to split a service than to merge several services back together, or to move a capability from one service to another. This is another reason why we recommend starting with a coarse-grained design and waiting until we learn more about the domain and have enough complexity before we split and increase service granularity.

We have found that there are three principles that work well together when thinking about the granularity of microservices. We call these principles the Universal Sizing Formula for microservices.

The Universal Sizing Formula

To achieve a reasonable sizing of microservices, you should:

- Start with just a few microservices, possibly using bounded contexts.
- Keep splitting as your application and services grow, being guided by the needs of coordination avoidance.
- Be on the right trajectory for decreasing coordination. This is vastly more important than the current state of how "perfectly" you get service sizing.

Summary

In this chapter we addressed a critical question of how to properly size microservices head-on. We looked at Domain-Driven Design, a popular methodology for modeling decomposition in complex systems; explained the process of conducting a highly efficient domain analysis with the Event Storming methodology, and introduced the Universal Sizing Formula, which offers unique guidance for effective sizing of microservices.

In the following chapters we will go deeper into implementation, showing how to manage data in a loosely coupled, componentized microservices environment. We also will walk you through a sample implementation for our demo project: an online reservation system.

CHAPTER 7

Building a Microservices Infrastructure

In the previous chapter we built a CI/CD pipeline for infrastructure changes. The infrastructure for our microservices system will be defined in code and we'll be able to use the pipeline to automate the testing and implementation of that code. With our automated pipeline in place, we can start writing the code that will define the infrastructure for our microservices-based application. That's what we'll focus on in this chapter.

Setting up the right infrastructure is vital to getting the most out of your microservices system. Microservices give us a nice way of breaking the parts of our application into bite-sized pieces. But we'll need a lot of supporting infrastructure to make all those bite-sized services work together properly. Before we can tackle the challenges of designing and engineering the services themselves, we'll need to spend some time establishing a network architecture and a deployment architecture for the services to use.

By the end of this chapter, you'll have built a cloud-based infrastructure designed to host the microservices we'll be building in the next chapter. We'll start by introducing the infrastructure and its components.

Infrastructure Components

The infrastructure is the set of components that will allow us to deploy, manage, and support a microservices-based application. An infrastructure can include a lot of parts: hardware, software, networks, and tools. So the scope of components we'll need to set up is quite large and getting all of those parts up and running is a big task.

Thankfully, as microservices approaches have matured, there's been an explosion in tools and services that make this work easier. In our model, we'll use tools like these as much as we can. We'll also focus on getting the infrastructure to work with a single

cloud platform (AWS) rather than building a cloud-agnostic application that can be "lifted and shifted" to other hosts. These decisions will make it possible for us to define a feature-rich infrastructure in the small space of this single chapter.

But it's still quite a challenge! We'll be covering a lot of topics in a small number of pages. That means we'll need to make some trade-offs. For example, we won't be able to cover security, operations controls, or event logging and support. Instead we'll focus on designing and writing Terraform code to create a working network, an AWS managed Kubernetes service, and a declarative GitOps server. These three components will give us the foundation we need to deploy our example microservices.

Network design and Kubernetes are deep and complex topics that require much more discussion than we can afford to give them. The good news is, you don't need to be a network or Kubernetes expert to set up your first microservices environment. If these are new domains for you, you can follow the instructions we've provided to get those parts of the system up and running as a first step to learning more about them.

Let's kick things off by taking a quick tour of our main components, starting with the network.

The Network

Microservices need to be run on a network. So we'll need to make sure we have a suitable one set up. Since we've made the decision to host our services on an AWS cloud, we'll need to tailor our network design accordingly, and create a virtual network instead of a physical one. We won't need to worry about the details of physical routers, cables, or network devices. Instead, we'll need to learn to use the language of AWS network resources and configure those accordingly.

We're going to keep our network design as simple as we can. We'll build just enough to get our system up and running. But, we'll still need to build and configure a few basic resources to support the running of our future microservices:

A virtual private cloud
: In AWS, a *virtual private cloud* (VPC) is the parent object for a virtual network. We'll be creating and configuring a VPC as part of our network design.

Subnets
: A VPC can be partitioned into multiple smaller networks called subnets. Subnets give us a way of organizing network traffic and controlling access to resources. We'll be creating a total of four subnets as part of our network configuration.

Routing and security
: In addition to creating the VPC and subnet objects, we'll be defining objects that dictate how traffic can flow in and out of them. For example, we'll be defining two "private" subnets that will only accept traffic from inside our VPC.

As you can see, our network has a bit of complexity that we'll need to deal with, including managing four "subnets" and connecting them. The main driver for this complexity comes from the needs of the Kubernetes service running on top of it. So, let's take a look at that next.

The Kubernetes Service

Throughout this book, we've emphasized that reducing coordination costs is an important success factor for the system. We've also mentioned containers and containerization a few times in earlier chapters. That's because containers are a great way of helping our teams get more done with less coordination costs. Containers give us the advantages of running applications in a predictable, isolated system configuration without the overhead and heavy lifting that comes with a VM deployment. Microservices and containers are a natural fit.

If you need help understanding containers and containerization, Docker's website has a nice introductory explanation of containers.

Containers make it easy for us to build microservices that run predictably across environments as a self-contained unit. But, containers don't know how to start themselves, scale themselves, or heal themselves when they break. Containers work great in isolation, but a lot of operations work is required to manage them in production-like environments. That's where Kubernetes comes in.

Kubernetes is a container orchestration tool developed by Google. It solves the problems of working with containers at scale. Kubernetes provides a tool-based solution for deploying, scaling, observing, and managing container-based applications. It can help you roll out and roll back container deployments, automatically deploy or destroy container deployments based on demand patterns, mount storage systems, manage secrets, and help with load balancing and traffic management. Kubernetes can do a lot of complicated and complex work, and has quickly become an essential part of a microservices infrastructure stack.

Kubernetes is also pretty complicated itself, however. Because of this, we won't be diving into the details of how Kubernetes works in this book. But we will be able to put together a working Kubernetes infrastructure hosted on AWS.

If you want to learn about Kubernetes, a great introduction is provided by *Kubernetes: Up and Running* by Brendan Burns, Joe Beda, and Kelsey Hightower (O'Reilly).

If you're new to Kubernetes, it's worth understanding the big moving parts in a Kubernetes system, so you'll be able to follow along as we set one up:

Kubernetes cluster
: The cluster is the parent object in a Kubernetes system. When you install Kubernetes, you are installing a cluster. A cluster contains a control plane and a set of nodes.

Control plane
: In Kubernetes, the control plane is the "brains" of the cluster. It manages the system by making decisions about starting, stopping, and replicating containers. The control plane also provides an API that we can use to administrate the cluster.

Nodes
: The runtime work happens in the nodes. Each node is a physical or virtual machine that runs the container-based workload. In Kubernetes, nodes run *pods*. Each pod contains one or more containers. Every cluster has at least one node.

In our implementation, we'll be using an AWS managed service for Kubernetes called Elastic Kubernetes Service (EKS). We're using EKS because it handles for us a lot of Kubernetes' complexity. It will help us provision the cluster and give us a control plane to use as a managed service. All we'll need to do is configure the number and types of nodes we want and provision a suitable network.

Key Decision: Use a Managed Kubernetes Service

We will use AWS EKS as a managed service for our Kubernetes cluster.

The last piece of our infrastructure is the GitOps deployment server. Let's find out more about what that is and how it will help us.

The GitOps Deployment Server

In Chapter 2, we introduced the release team. This is the team that will be responsible for deploying microservices into production. We expect our microservices teams to use a CI/CD pipeline to integrate, test, build, and deliver their services. But in our operating model they don't own the actual deployment of the service into the system. We made that decision because in our experience, production deployments are fairly

complex and require special attention. To facilitate the deployment work of the release team, we're introducing a special tool in our platform service offering: a GitOps deployment server.

Continuous Delivery Versus Continuous Deployment

One of the confusing things about CI/CD is that the "CD" part can mean either Continuous Delivery or Continuous Deployment, depending on who you ask. In our model, Continuous Delivery (CD) of microservices happens when our teams are able to automatically and continually ship their finished microservices as containers. Deployment happens when these containers are released into the production environment by our release team.

The name *GitOps*, created by a company called WeaveWorks, describes a way of working that uses Git as a "source of truth." That means that whatever is in Git should be the target state for the system. Like Terraform, GitOps prescribes a declarative approach. GitOps tools need to do the work of synchronizing system configurations to look like the state described in the Git repository. They also need to alert operations teams if the real world has drifted from the state defined in Git.

Argo CD is a GitOps tool that facilitates the work of deploying Kubernetes applications. We've decided to use Argo CD for our release process because we like the declarative GitOps approach. If we didn't use Argo CD we'd need to automate a series of Kubernetes calls to deploy an application. Instead, with the GitOps approach, we only need to point Argo CD at a Git source and let it do the work of keeping our environment up to date.

For example, once we have Argo CD set up, we'll be able to have it watch a microservice code repository. When new changes are comitted and tested, Argo CD will be able to automatically deploy the new version of the service into the environment. This declarative, continuous deployment capability makes Argo CD a great product for our release teams to use.

Key Decision: Deploy Microservices Using a GitOps Deployment Tool

Our release teams will use Argo CD to manage microservice deployment into production and production-like environments.

By the end of this chapter, we'll have created a sandbox environment with an Argo CD server installed on top of an AWS managed Kubernetes cluster, running on an AWS VPC network. It will take a lot of Terraform code to make that happen, so prepare to roll up your sleeves. We'll dive into the build in the next section.

Implementing the Infrastructure

In Chapter 6 we established a decision to use Terraform to write the code that defines our infrastructure and GitHub Actions to test and apply our infrastructure changes. In this section, we'll break our infrastructure design into discrete Terraform modules and call them from the sandbox environment we started building in the previous chapter. We'll start by setting up the tools you'll need in your infrastructure development workspace.

Installing kubectl

If you've followed along with the instructions in Chapter 6, you'll already have an environment ready for the infrastructure build. So you'll have:

- An AWS instance and a configured operator account
- Git, Terraform, and AWS CLI tools installed in your workstation
- A GitHub Actions pipeline for the infrastructure

If you haven't yet set up your GitHub Actions pipeline, or you had trouble getting it to work the way we've described, you can create a fork of a basic sandbox environment by following the instructions in this book's GitHub repository.

In addition to the setup we've done in the previous chapter, you'll need to do one more installation step to get ready for this chapter: installing `kubectl`. When we're installing the Kubernetes service, we'll need a way to test and interact with the Kubernetes system. To do that, we'll use the command-line application `kubectl` to interact with a Kubernetes server.

Follow the instructions in the Kubernetes documentation to install `kubectl` in your local system. We'll leave it to you to pick the flavor that's appropriate to your operating system.

With the workspace set up and ready to go, we can move on to writing the Terraform modules that will define the infrastructure.

Setting Up the Module Repositories

When you write professional software, it's important to write clean, professional code. When code is too difficult to understand, to maintain, or to change, the project becomes costly to operate and maintain. All of that is true for our infrastructure code as well.

Since we're taking the IaC approach, we'll need to apply good code practices to our infrastructure project. The good news is that we have lots of existing guidance in our industry on how to write code that is easier to learn, understand, and extend. The bad

news is that not every principle and practice from traditional software development is going to be easy to implement in the IaC domain. That's partly because the tooling and languages for IaC are still evolving and partly because the context of changing a live, physical device is a different model from the traditional software development model.

But with Terraform we'll be able to apply three essential coding practices that will help us write clean, easier-to-maintain code:

Use modules
: Writing small functions that do one thing well

Encapsulate
: Hiding internal data structures and implementation details

Avoid repetition
: Don't repeat yourself (DRY), implementing code once in only one location

Terraform's built-in support for modules of infrastructure code will help us in using all three of these practices. We'll be able to maintain our infrastructure code as a set of reusable, encapsulated modules. We'll build modules for each of the architecturally significant parts of our system: networks, the API gateway, and the managed Amazon Kubernetes service (EKS). Once we have our reusable modules in place, we'll be able to implement another set of Terraform files that use them. We'll be able to have a different Terraform file for each environment that we want to create without repeating the same infrastructure declarations in each one (see Figure 7-1).

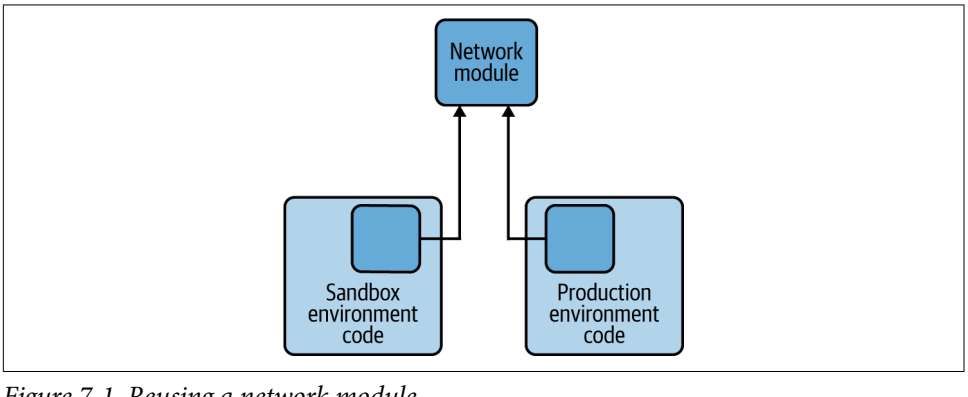


Figure 7-1. Reusing a network module

This approach allows us to easily spin up new environments by creating new Terraform files that reuse the modules we've developed. It also lets us make changes in just one place when we want to change an infrastructure configuration across all environments. We can start by creating a simple module that defines a basic network and an environment file that uses it.

The infrastructure code we are writing in this chapter uses Terraform's module structure. Each module will have its own directory and contain *variables.tf*, *main.tf*, and *output.tf* files. The advantage of this approach is that you can define a module once and use it in a parameterized way to build multiple environments. You can learn more about these modules in the Terraform documentation.

We're going to create two modules for our microservices infrastructure. First, we'll create an AWS networking module that contains a declarative configuration of our software-defined network. We'll also create a Kubernetes module that defines an AWS-based managed Kubernetes configuration for our environments. We'll be able to use both of these modules to create our sandbox environment.

Don't Use Our Configuration Files in Your Production Environment!

We've done our best to design an infrastructure that mirrors production environments that large organizations use for microservices. But space constraints prevent us from giving you a comprehensive set of configurations that will work for your specific environment, security needs, and constraints. You can use this chapter as a quick starter and guide to the tools you'll need, but we advise that you spend time designing your own production-grade infrastructure, configuration, and architecture.

In Chapter 6, we created a GitHub code repository for the sandbox environment code and its CI/CD pipeline. We'll be using that code repository in this chapter, but we'll also create a new repository for each module we write. Terraform has built-in support for importing modules that are managed as GitHub repositories, so it will be easy to pull them in when we want to use them.

To get started, let's create the repositories for all the modules we'll be writing in this chapter. Go ahead and create three new public GitHub-hosted repositories with the names described in Table 7-1.

Table 7-1. Infrastructure module names

Repository name	Visibility	Description
module-aws-network	Public	A Terraform module that creates the network
module-aws-kubernetes	Public	A Terraform module that sets up EKS
module-argo-cd	Public	A Terraform module that installs Argo CD into a cluster

If you aren't sure how to create a GitHub repository, you can follow the GitHub instructions.

We recommend that you make these repositories public so that they are easier to import into your Terraform environment definition. You can use private repositories if you prefer—you'll just have to add some authentication information to your import command so that Terraform can get to the files correctly. You should also add a *.gitignore* file to these repositories so you don't end up with a lot of Terraform working files pushed to your GitHub server. You can do that by choosing a Terraform *.gitignore* in the GitHub web GUI, or save the contents as a *.gitignore* file in the root directory of your code repository, as outlined on this GitHub site.

With our three GitHub module repositories created and ready to be populated, we can dive into the work of actually writing the actual infrastructure definitions—starting with the network.

The Network Module

The virtual network is a foundational part of our infrastructure, so it makes sense for us to start by defining the network module. In this section, we'll write an AWS network module that will support a specific Kubernetes and microservices architecture and workload. Because it's a module, we'll be writing input, main, and output code—just like we'd write inputs, logic, and return values for an application function. When we're done, we'll be able to use this module to easily provision a network environment by specifying just a few input values.

We'll be writing the network infrastructure code in the `module-aws-network` GitHub repository that you created earlier. We'll be creating and editing Terraform files in the root directory of this module. If you haven't already done so, clone the repository into your local environment and get your favorite text editor ready.

A completed listing for this AWS network module is available in this book's GitHub repository.

Network module outputs

Let's start by defining the resources that we expect the networking module to produce. We'll do this by creating a Terraform file called *output.tf* in the root directory of module-aws-network, as in Example 7-1.

Example 7-1. module-aws-network/output.tf

```
output "vpc_id" {
  value = aws_vpc.main.id
}

output "subnet_ids" {
  value = [
    aws_subnet.public-subnet-a.id,
    aws_subnet.public-subnet-b.id,
    aws_subnet.private-subnet-a.id,
    aws_subnet.private-subnet-b.id]
}

output "public_subnet_ids" {
  value = [aws_subnet.public-subnet-a.id, aws_subnet.public-subnet-b.id]
}

output "private_subnet_ids" {
  value = [aws_subnet.private-subnet-a.id, aws_subnet.private-subnet-b.id]
}
```

Based on the Terraform module output file, we can see that the network module creates a VPC resource that represents the software-defined network for our system. Within that network, our module will also create four logical subnets—these are the bounded parts of our network (or subnetworks). Two of these subnets will be public, meaning that they will be accessible over the internet. Later, we'll use all four subnets for our Kubernetes cluster setup and eventually we'll deploy our microservices into them.

Network module main configuration

With the output of our module defined, we can start putting together the declarative code that builds it and creates the outputs we are expecting. In a Terraform module, we'll be creating and editing a file named *main.tf* in the root directory of the `module-aws-network` repository.

Getting the Source Code

To help you understand the network implementation, we've broken the *main.tf* source code file into smaller parts. You can find the complete source code listing for this module at this book's GitHub site.

We'll start our module implementation by creating an AWS VPC resource. Terraform provides us with a special resource for defining AWS VPCs, so we'll just need to fill in a few parameters to create our definition. When we create a resource in Terraform,

we define the parameters and configuration details for it in the Terraform syntax. When we apply these changes, Terraform will make an AWS API call and create the resource if it doesn't exist already.

You can find all the Terraform documentation for the AWS provider on the Terraform site. You can also consult this documentation if you're building a similar implementation in GCP or Azure.

Create a file called *main.tf* in the root of your network module's repository and add the Terraform code in Example 7-2 to the *main.tf* file to define a new AWS VPC resource.

Example 7-2. modules-aws-network/main.tf

```
provider "aws" {
  region = var.aws_region
}

locals {
  vpc_name = "${var.env_name} ${var.vpc_name}"
  cluster_name = "${var.cluster_name}-${var.env_name}"
}


## AWS VPC definition
resource "aws_vpc" "main" {
  cidr_block = var.main_vpc_cidr
  tags = {
    "Name"                                          = local.vpc_name,
    "kubernetes.io/cluster/${local.cluster_name}" = "shared",
  }
}
```

The network module starts with a declaration that it is using the AWS *provider*. This is a special instruction that lets Terraform know that it needs to download and install the libraries it will need in order to communicate with the AWS API and create resources on our behalf. When we validate or apply this file in Terraform, it will attempt to connect to the AWS API using the credentials we've configured in the system as environment variables. We're also specifying an AWS region here so that Terraform knows which region it should be working in.

We've also specified two local variables using a Terraform `locals` block. These variables define a naming standard that will help us differentiate environment resources in the AWS console. This is especially important if we plan to create multiple environments in the same AWS account space as it will help us avoid naming collisions.

After the local variable declaration, you'll find the code for creating a new AWS VPC. As you can see, there isn't much to it, but it does define two important things: a CIDR block and a set of descriptive tags.

Classless inter-domain routing (CIDR) is a standard way of describing an IP address range for the network. It's a shorthand string that defines which IP addresses are allowed inside a network or a subnet. For example, a CIDR value of `10.0.0.0/16` would mean that you could bind to any IP address between 10.0.0.0 and 10.0.255.255 inside the VPC. We'll be defining a pretty standard CIDR range for you when we build the sandbox environment, but for more details on how CIDRs work and why they exist, you can read about them in the RFC.

We've also added some tag values to the VPC. Resource tags are useful because they give us a way of easily identifying groups of resources when we need to administrate them. Tags are also useful for automated tasks and for identifying resources that should be managed in specific ways. In our definition, we have defined a "Name" tag to make our VPC easier to identify. We've also defined a Kubernetes tag that identifies this cluster as a target for our Kubernetes cluster (which we'll define in "Defining the EKS cluster" on page 64).

Also, notice that in a few cases we've referenced a variable instead of an actual value in our configuration. For example, our CIDR block is defined as `var.main_vpc_cidr` and it has a Name tag with the value `local.vpc_name`. These are Terraform variables, and we'll define their values later when we use this module as part of our sandbox environment. The variables are what makes the modules reusable—by changing the variable values we can change the types of environments that we create.

With our main VPC defined, we can move on to configuring the subnets for the network. As we mentioned earlier in this chapter, we'll be using Amazon's managed Kubernetes service (EKS) to run our workload. In order for EKS to function properly, we'll need to have subnets defined in two different "availability zones." In AWS, an availability zone represents a separate physical data center. It's a useful construct, because even though the AWS resources are virtual, they're still running on a computer plugged into an outlet somewhere. By using two availability zones for our deployment, we ensure that our services will still work even if one of the data centers goes down.

In addition to configuring two availability zones, Amazon also recommends a VPC configuration with both public and private subnets. So our network will have public subnets that allow traffic from the internet and private subnets that will only allow traffic from inside the VPC. When EKS is running, it will deploy load balancers in the public subnet to manage inbound traffic, which will be routed to our containerized microservices deployed in the private subnets.

To meet those requirements, we'll define a total of four subnets. Two of them will be designated as public subnets, so they'll be accessible over the web. The other two subnets will be private. We'll also split our public and private subnets up so that they are deployed in separate availability zones. When we're done, we'll have a network that looks like Figure 7-2.

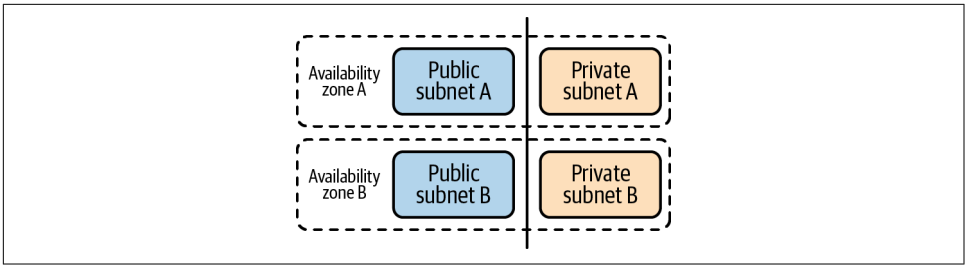


Figure 7-2. AWS subnet design

We've already specified a CIDR for the IP range in our VPC. Now we'll need to split up those IP addresses for the subnets to use. Since the subnets are inside of the VPC, they'll need to have a CIDR that is within the boundaries of the VPC IP range. We won't actually be defining those IP addresses in our module though. Instead, we'll use variables just like we did for the VPC.

In addition to the CIDR blocks, we'll specify the availability zones for our subnets as a parameter. Rather than hardcoding the name of the availability zone, we'll use a special Terraform type called `data` that will let us dynamically choose the zone name. In this case, we'll put `public-subnet-a` and `private-subnet-a` in `data.aws.availability_zones.available.names[0]` and `public-subnet-b` and `private-subnet-b` in `data.aws.availability_zones.available.names[1]`. Using dynamic data like this makes it easier for us to spin up this infrastructure in different regions.

Finally, we'll add a name tag so that we can easily find our network resources through the admin and ops consoles. We'll also need to add some EKS tags to the subnet resources so that our AWS Kubernetes service will know which subnets we are using and what they are for. We'll tag our public subnets with an `elb` role so that EKS knows it can use these subnets to create and deploy an elastic load balancer. We'll tag the private subnets with an `internal-elb` role to indicate that our workloads will be deployed into them and can be load balanced. For more details on how AWS EKS uses load balancer tags, consult the AWS documentation.

Add the Terraform code in Example 7-3 to the end of your *main.tf* file in order to declare the subnet configuration.

Example 7-3. modules-aws-network/main.tf (subnets)

```
# subnet definition

data "aws_availability_zones" "available" {
  state = "available"
}

resource "aws_subnet" "public-subnet-a" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.public_subnet_a_cidr
  availability_zone = data.aws_availability_zones.available.names[0]

  tags = {
    "Name"                                      = (
      "${local.vpc_name}-public-subnet-a"
    )
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/elb"                    = "1"
  }
}

resource "aws_subnet" "public-subnet-b" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.public_subnet_b_cidr
  availability_zone = data.aws_availability_zones.available.names[1]

  tags = {
    "Name"                                      = (
      "${local.vpc_name}-public-subnet-b"
    )
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/elb"                    = "1"
  }
}

resource "aws_subnet" "private-subnet-a" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.private_subnet_a_cidr
  availability_zone = data.aws_availability_zones.available.names[0]

  tags = {
    "Name"                                      = (
      "${local.vpc_name}-private-subnet-a"
    )
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/internal-elb"           = "1"
  }
}

resource "aws_subnet" "private-subnet-b" {
  vpc_id            = aws_vpc.main.id
```

```
  cidr_block        = var.private_subnet_b_cidr
  availability_zone = data.aws_availability_zones.available.names[1]

  tags = {
    "Name"                                          = (
      "${local.vpc_name}-private-subnet-b"
    )
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/internal-elb"             = "1"
  }
}
```

In Terraform, a `data` element is a way of querying the provider for information. In the network module, we're using the `aws_availability_zones` data element to ask AWS for availability zone IDs in the region we've specified. This is a nice way to avoid hardcoding values into the module.

Although we've configured four subnets and their IP ranges, we haven't yet defined the network rules that AWS will need to manage traffic through them. To finish our network design, we'll need to implement a set of routing tables that define what traffic sources we will allow into our subnets. For example, we'll need to establish how traffic will be routed through our public subnets and how each of the subnets will be allowed to communicate with each other.

We'll start by defining the routing rules for our two public subnets: `public-subnet-a` and `public-subnet-b`. To make these subnets accessible on the internet, we'll need to add a special resource to our VPC called an *internet gateway*. This is an AWS network component that connects our private cloud to the public internet. Terraform gives us a resource definition for the gateway, so we'll use that and tie it to our VPC with the `vpc_id` configuration parameter.

Once we've added the internet gateway, we'll need to define routing rules that let AWS know how to route traffic from the gateway into our subnets. To do that, we'll create an `aws_route_table` resource that allows all traffic from the internet (which we'll identify with CIDR block 0.0.0/0) through the gateway. Then we just need to create associations between our two public subnets and the table we've defined.

Add the Terraform code in Example 7-4 to *main.tf* to define routing instructions for our network.

Example 7-4. modules-aws-network/main.tf (public routes)

```
# Internet gateway and routing tables for public subnets
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main.id

  tags = {
    Name = "${local.vpc_name}-igw"
  }
}

resource "aws_route_table" "public-route" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }

  tags = {
    "Name" = "${local.vpc_name}-public-route"
  }
}

resource "aws_route_table_association" "public-a-association" {
  subnet_id      = aws_subnet.public-subnet-a.id
  route_table_id = aws_route_table.public-route.id
}

resource "aws_route_table_association" "public-b-association" {
  subnet_id      = aws_subnet.public-subnet-b.id
  route_table_id = aws_route_table.public-route.id
}
```

With the routes for our public subnets defined, we can dive into the setup for our two private subnets. The route configuration for the private subnets will be a bit more complicated than what we've done so far. That's because we'll need to define a route from our private subnet out to the internet to allow our Kubernetes Pods to talk to the EKS service.

For that kind of route to work, we'll need a way for nodes in our private subnet to talk to the internet gateway we've deployed in the public subnets. In AWS, we'll need to create a network address translation (NAT) gateway resource that gives us a path out. When we create the NAT, we'll also need to assign it a special kind of IP address called an *elastic IP address* (or EIP). Because this is an AWS construct, the IP is a real internet-accessible network address, unlike all the other addresses in our network, which are virtual and exist inside AWS alone. Since real IP addresses aren't unlimited, AWS limits the amount of these available. Unfortunately, we can't create an NAT without one, so we'll have to use two of them—one for each NAT we are creating.

Add the Terraform code in Example 7-5 to implement an NAT gateway in our network.

Example 7-5. modules-aws-network/main.tf (NAT gateway)

```
resource "aws_eip" "nat-a" {
  vpc = true
  tags = {
    "Name" = "${local.vpc_name}-NAT-a"
  }
}

resource "aws_eip" "nat-b" {
  vpc = true
  tags = {
    "Name" = "${local.vpc_name}-NAT-b"
  }
}

resource "aws_nat_gateway" "nat-gw-a" {
  allocation_id = aws_eip.nat-a.id
  subnet_id     = aws_subnet.public-subnet-a.id
  depends_on    = [aws_internet_gateway.igw]

  tags = {
    "Name" = "${local.vpc_name}-NAT-gw-a"
  }
}

resource "aws_nat_gateway" "nat-gw-b" {
  allocation_id = aws_eip.nat-b.id
  subnet_id     = aws_subnet.public-subnet-b.id
  depends_on    = [aws_internet_gateway.igw]

  tags = {
    "Name" = "${local.vpc_name}-NAT-gw-b"
  }
}
```

In addition to the NAT gateway we've created, we'll need to define routes for our private subnets. Add the Terraform code in Example 7-6 to *main.tf* to complete the definition of our network routes.

Example 7-6. modules/network/main.tf (private routes)

```
resource "aws_route_table" "private-route-a" {
  vpc_id = aws_vpc.main.id
  route {
    cidr_block     = "0.0.0.0/0"
    nat_gateway_id = aws_nat_gateway.nat-gw-a.id
```

```
  }

  tags = {
    "Name" = "${local.vpc_name}-private-route-a"
  }
}

resource "aws_route_table" "private-route-b" {
  vpc_id = aws_vpc.main.id
  route {
    cidr_block     = "0.0.0.0/0"
    nat_gateway_id = aws_nat_gateway.nat-gw-b.id
  }

  tags = {
    "Name" = "${local.vpc_name}-private-route-b"
  }
}

resource "aws_route_table_association" "private-a-association" {
  subnet_id      = aws_subnet.private-subnet-a.id
  route_table_id = aws_route_table.private-route-a.id
}

resource "aws_route_table_association" "private-b-association" {
  subnet_id      = aws_subnet.private-subnet-b.id
  route_table_id = aws_route_table.private-route-b.id
}
```

That's it for our main network definition. When we eventually run this Terraform file, we'll have an AWS software-defined network that is ready for Kubernetes and our microservices. But, before we can use it, we'll need to define all of the input variables that this module needs. Although we've referenced a lot of `var` values in our code, Terraform modules require us to identify all of the input variables we'll be using in a specific file called *variables.tf*. If we don't do that, we won't be able to pass variable values into our module.

Network module variables

Create a file in the root folder of the network module called *variables.tf*. Add the Terraform code in Example 7-7 to *variables.tf* to define the inputs for the module.

Example 7-7. modules/network/variables.tf

```
variable "env_name" {
  type = string
}

variable "aws_region" {
  type = string
}

variable "vpc_name" {
  type    = string
  default = "ms-up-running"
}

variable "main_vpc_cidr" {
  type = string
}

variable "public_subnet_a_cidr" {
  type = string
}

variable "public_subnet_b_cidr" {
  type = string
}

variable "private_subnet_a_cidr" {
  type = string
}

variable "private_subnet_b_cidr" {
  type = string
}

variable "cluster_name" {
  type = string
}
```

As you can see, the variable definitions are fairly self-explanatory. They describe a name, optional description, and type value. In our module we're only using string values. In some cases, we've also provided a default value so that those inputs don't always have to be defined for every environment. We'll give the module values for those variables when we use it to create an environment.

It's good practice to include a description attribute for every variable in your Terraform module. This improves the maintainability and usability of your modules and becomes increasingly important over time. We've done this for the Terraform files we've published in GitHub, but we've removed the descriptions in all our examples to save space in the book.

The Terraform code for our network module is now complete. At this point, you should have a list of files that looks something like this in your module directory:

```
drwxr-xr-x   3 msur  staff   96 14 Jun 09:57 ..
drwxr-xr-x   7 msur  staff  224 14 Jun 09:58 .
-rw-r--r--   1 msur  staff   23 14 Jun 09:57 README.md
drwxr-xr-x  13 msur  staff  416 14 Jun 09:57 .git
-rw-r--r--   1 msur  staff    0 14 Jun 09:58 main.tf
-rw-r--r--   1 msur  staff  612 14 Jun 09:58 variables.tf
-rw-r--r--   1 msur  staff   72 14 Jun 09:58 outputs.tf
```

With the code written, we'll be testing the network module by creating a sandbox environment network, but before we use the module we should make sure we haven't made any syntax errors. The Terraform command-line application includes some handy features to format and validate code. If you haven't already installed the Terraform client in your local system, you can find a binary for your operating system on the Terraform site.

Use the following Terraform command while you are in your module's working directory to format the module's code:

```
module-aws-network$ terraform fmt
```

The `fmt` command will *lint*, or format, all the Terraform code in the working directory and ensure that it conforms to a set of built-in style guidelines. It will automatically make those changes for you and will list any files that have been updated.

Next, run `terraform init` so that Terraform can install the AWS provider libraries. We need to do this so that we can validate the code. Note that you'll need to have AWS credentials defined for this to work. If you haven't done that yet, follow the instructions in the previous chapter:

```
module-aws-network$ terraform init
```

If you run into any problems, try to fix those before you continue; the Terraform documentation has a helpful section on troubleshooting. Finally, you can run the `validate` command to make sure that our module is syntactically correct:

```
module-aws-network$ terraform validate
Success! The configuration is valid.
```

If you need to debug your Terraform code, you can set the environment variable `TF_LOG` to `INFO` or `DEBUG`. That will instruct Terraform to emit logging info to standard output.

When you are satisfied that the code is formatted and valid, you can commit your changes to the GitHub repository. If you've been working in a local repository, you can use the following command to push your changes to the main repository:

```
module-aws-network$ git add .
module-aws-network$ git commit -m "network module created"
[master ddb7e41] network module created
 3 files changed, 226 insertions(+)
module-aws-network$ git push
```

Our Terraform-based network module is now complete and available for use. It has a *variables.tf* file that describes the required and optional input variables to use it. It has a *main.tf* file that declaratively defines the resources for our network design. Finally, it has an *outputs.tf* file that defines the significant resources that we've created in the module. Now we can use the module to create a real network in our sandbox environment.

Create a sandbox network

The nice thing about using Terraform modules is that we can create our environments easily in a repeatable way. Outside of the specific values we've defined in the *variables.tf* file, any environment that we create with the module we've defined will operate with a network infrastructure that we know and understand. That means we can expect our microservices to work in a predictable way as we move them through testing and release environments since we have reduced the level of variation.

But to apply the module we've defined and create a new environment, we'll need to call it from a Terraform file that defines values for the module's variables. To do that, we'll create a sandbox environment that demonstrates a practical example of using a Terraform module. If you followed the steps in Chapter 6, you'll already have a code repository for your sandbox environment with a single *main.tf* file in it.

In order to use the network module that we've created, we'll use a special Terraform resource called `module`. It allows us to reference a Terraform module that we've created and pass in values for the variables that we've defined. Terraform expects a property called `source` to exist in the module that indicates where it can find the code.

In our case, we want Terraform to retrieve the network module from a GitHub repository. To do this, we need to use a `source` property that starts with the string `"github.com"` and contains the path of our repository. That lets Terraform know it needs to pull the source from GitHub.

For example, a source value of "`github.com/implementing-microservices/module-aws-network`" references our example network module. You can find the path for your module's repository by copying the path from its GitHub URL (see Table 7-2).

Table 7-2. Sandbox environment network variable

Name	Description	Example
YOUR_NETWORK_MODULE_REPO_PATH	The path to your module's repository in GitHub	github.com/implementing-microservices/module-aws-network

When you have the path for your network module ready, open the *main.tf* file for the sandbox environment you created in Chapter 6. Add the Terraform code in Example 7-8 after the `# Network Configuration` comment. Don't forget to replace the `source` value with the path of your network module's GitHub repository.

Example 7-8. env-sandbox/main.tf (network)

```
...

# Network Configuration
module "aws-network" {
  source = "github.com/{YOUR_NETWORK_MODULE_REPO_PATH}"

  env_name              = local.env_name
  vpc_name              = "msur-VPC"
  cluster_name          = local.k8s_cluster_name
  aws_region            = local.aws_region
  main_vpc_cidr         = "10.10.0.0/16"
  public_subnet_a_cidr  = "10.10.0.0/18"
  public_subnet_b_cidr  = "10.10.64.0/18"
  private_subnet_a_cidr = "10.10.128.0/18"
  private_subnet_b_cidr = "10.10.192.0/18"
}

# EKS Configuration

# GitOps Configuration
```

Amazon's S3 bucket names must be globally unique, so you'll need to change the value of `bucket` to something that is unique and meaningful for you. Refer to Chapter 6 for instructions on how to set up the backend. If you want to do a quick and dirty test, omit the backend definition and Terraform will store state locally in your filesystem.

Our infrastructure pipeline will apply Terraform changes, but before we kick it off we need to check to make sure that the Terraform code we've written will work. A good first step is to format and validate the code locally:

```
$ terraform fmt
[...]
$ terraform init
[...]
$ terraform validate
Success! The configuration is valid.
```

If you need to debug the networking module and end up making code changes, you may need to run the following command in your sandbox environment directory:

```
$ terraform get -update
```

This will instruct Terraform to pull the latest version of the network module from GitHub.

If the code is valid, we can get a plan to validate the changes that Terraform will make when they are applied. It's always a good idea to do a dry run and examine the changes that will be made before you actually change the environment, so make this step a part of your workflow. To get the Terraform plan, run this command:

```
$ terraform plan
```

Terraform will provide you with a list of the resources that will be created, deleted, and updated. If Terraform and AWS are new to you, the plan might be difficult to evaluate and understand in detail. But, you should still be able to get a general sense of what is going to happen. Since this is the first update, the plan should list a lot of new resources that Terraform will create. When you're ready, you can push the code to the GitHub repository and `tag` it for release:

```
$ git add .
$ git commit -m "initial network release"
$ git push origin
$ git tag -a v1.0 -m "network build"
$ git push origin v1.0
```

There are two `git push` commands that we need to run. The first one pushes the code changes we've made and the second only pushes the tag.

With the code tagged and pushed, our GitHub Actions pipeline should take over and start building the network for our sandbox environment. You'll need to log in to GitHub and check the Actions tab in your sandbox environment repository to make sure

that everything goes according to plan. If you don't remember how to do that, you'll find instructions in Chapter 6.

You can test that the VPC has been successfully created by making an AWS CLI call. Run the following command to list the VPCs with a CIDR block that matches the one that we've defined:

```
$ aws ec2 describe-vpcs --filters Name=cidr,Values=10.10.0.0/16
```

You should get a JSON body back describing the VPC that we created. If that has happened, it indicates that you now have an AWS network running and ready to use. It's now time to start writing the module for the Kubernetes service.

The Kubernetes Module

One of the most important parts of our microservices infrastructure is the Kubernetes layer that orchestrates our container-based services. If we set it up correctly, Kubernetes will give us an automated solution for resiliency, scaling, and fault tolerance. It will also give us a great foundation for deploying our services in a dependable way. On top of that, an Istio service mesh gives us a powerful way of managing traffic and improving the way our microservices communicate.

To build our Kubernetes module, we'll follow the same steps that we did to build our network module. We'll start by defining a set of output variables that define what the module will produce, then we'll write the code that declaratively defines the configuration that Terraform will create. Finally, we'll define the inputs. As we mentioned earlier in this chapter, we are managing each of infrastructure modules in it's own GitHub code repository. So make sure you start by creating a new GitHub repository for our Kubernetes module if you haven't done so already.

Implementing Kubernetes can get very complicated. So, to get our system up and running as quickly as possible, we'll use a managed service that will hide some of the setup and management complexity for us. Since we are running on AWS in our examples, we'll use the EKS bundled in Amazon's cloud offering.

The configuration for managed Kubernetes services tends to be very vendor specific, so the examples we provide here will likely take some reworking if you want to use them in Google Cloud, Azure, or another hosted service.

An EKS cluster contains two parts: a control plane that hosts the Kubernetes system software and a node group that hosts the VMs that our microservices will run on. In order to configure EKS, we'll need to provide parameters for both of these areas. When the module is finished running, we can return an EKS cluster identifier so that we have the option of inspecting or adding to the cluster with other modules.

With all that in mind, let's dive into the code that will make it come to life. We'll be working in the `module-aws-kubernetes` GitHub repository that you created earlier, so make sure you start by cloning it to your local machine. When you've done that, we can begin by editing the Terraform outputs file.

A completed listing for this Kubernetes module is available in this book's GitHub repository.

Kubernetes module outputs

We'll start by declaring the outputs that our module provides. Create a Terraform file called *outputs.tf* in the root directory of the `module-aws-kubernets` repository and add to it the code in Example 7-9.

Example 7-9. module-aws-kubernetes/outputs.tf

```
output "eks_cluster_id" {
  value = aws_eks_cluster.ms-up-running.id
}

output "eks_cluster_name" {
  value = aws_eks_cluster.ms-up-running.name
}

output "eks_cluster_certificate_data" {
  value = aws_eks_cluster.ms-up-running.certificate_authority.0.data
}

output "eks_cluster_endpoint" {
  value = aws_eks_cluster.ms-up-running.endpoint
}

output "eks_cluster_nodegroup_id" {
  value = aws_eks_node_group.ms-node-group.id
}
```

The main value we're returning is the identifier for the EKS cluster that we'll be creating in this module. The rest of the values need to be returned so that we can access the cluster from other modules once the cluster is ready and operational. For example, we'll need the endpoint and certificate data when we install the Argo CD server into this EKS cluster at the end of the chapter.

While the output of our module is pretty simple, the work of getting our EKS-based Kubernetes system set up is going to be a bit more complicated. Just like we did

before, we'll build the module's main Terraform file in parts before we test it and apply it.

Defining the EKS cluster

To start, create a Terraform file called *main.tf* in the root directory of your Kubernetes module and add an AWS provider definition, as in Example 7-10.

Example 7-10. module-aws-kubernetes/main.tf

```
provider "aws" {
  region = var.aws_region
}
```

Remember that we'll be using the Terraform naming convention of `var` to indicate values that can be replaced by variables when our module is invoked.

As we mentioned earlier, we're going to use Amazon's EKS to create and manage our Kubernetes installation. But EKS will need to create and modify AWS resources on our behalf in order to run. So we'll need to set up permissions in our AWS account so that it can do the work it needs to do. We'll need to define policies and security rules at the overall cluster level and also for the VMs or nodes that EKS will be spinning up for us to run microservices on.

We'll start by focusing on the rules and policies for the entire EKS cluster. Add the Terraform code in Example 7-11 to your *main.tf* file to define a new cluster access management policy.

Example 7-11. module-aws-kubernetes/main.tf (cluster access management)

```
locals {
  cluster_name = "${var.cluster_name}-${var.env_name}"
}

resource "aws_iam_role" "ms-cluster" {
  name = local.cluster_name

  assume_role_policy = <<POLICY
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "eks.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
```

```
  ]
}
POLICY
}

resource "aws_iam_role_policy_attachment" "ms-cluster-AmazonEKSClusterPolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"
  role       = aws_iam_role.ms-cluster.name
}
```

The snippet here establishes a trust policy that allows the AWS EKS service to act on your behalf. It defines a new identity and access management role for our EKS service and attaches a policy called `AmazonEKSClusterPolicy` to it. This policy has been defined by AWS for us and gives the EKS the permissions it needs to create VMs and make network changes as part of its Kubernetes management work. Notice that we are also defining and using a local variable for the name of the cluster. We'll use that variable throughout the module.

Now that the cluster service's role and policy are defined, add the code in Example 7-12 to your module's *main.tf* file to define a network security policy for the cluster.

Example 7-12. module-aws-kubernetes/main.tf (network security policy)

```
resource "aws_security_group" "ms-cluster" {
  name        = local.cluster
  vpc_id      = var.vpc_id

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "ms-up-running"
  }
}
```

A VPC security group restricts the kind of traffic that can go into and out of the network. The Terraform code we've just written defines an egress rule that allows unrestricted outbound traffic, but doesn't allow any inbound traffic, because there is no ingress rule defined. Notice that we are applying this security group to a VPC that will be defined by an input variable. When we use this module, we can give it the ID of the VPC that our networking module has created.

With these policies and a security group defined for the EKS cluster, we can now add the declaration for the cluster itself to the *main.tf* Terraform file (see Example 7-13).

Example 7-13. module-aws-kubernetes/main.tf (cluster definition)

```
resource "aws_eks_cluster" "ms-up-running" {
  name     = local.cluster_name
  role_arn = aws_iam_role.ms-cluster.arn

  vpc_config {
    security_group_ids = [aws_security_group.ms-cluster.id]
    subnet_ids         = var.cluster_subnet_ids
  }

  depends_on = [
    aws_iam_role_policy_attachment.ms-cluster-AmazonEKSClusterPolicy
  ]
}
```

The EKS cluster definition we've just created is pretty simple. It simply references the name, role, policy, and security group values we defined earlier. It also references a set of subnets that the cluster will be managing. These subnets will be the ones that we created earlier in the networking module, and we'll be able to pass them into this Kubernetes module as a variable.

When AWS creates an EKS cluster, it automatically sets up all of the management components that we need to run our Kubernetes cluster. This is called the *control plane* because it's the brain of our Kubernetes system. But in addition to the control plane, our microservices need a place where they can run. In Kubernetes, that means we need to set up nodes—the physical or VMs that containerized workloads can run on.

One of the advantages of using a managed Kubernetes service like EKS is that we can offload some of the work of managing the creation, removal, and updating of Kubernetes nodes. For our configuration, we'll define a managed EKS node group and let AWS provision resources and interact with the Kubernetes system for us. But to get a managed node group running, we'll still need to define a few important configuration values.

Defining the EKS node group

Just like we did for our cluster, we'll begin the node configuration by defining a role and some security policies. Add the node group IAM definitions in Example 7-14 to the Kubernetes module's *main.tf* file.

Example 7-14. module-aws-kubernetes/main.tf (node group IAM)

```
# Node Role
resource "aws_iam_role" "ms-node" {
  name = "${local.cluster_name}.node"

  assume_role_policy = <<POLICY
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
POLICY
}

# Node Policy
resource "aws_iam_role_policy_attachment" "ms-node-AmazonEKSWorkerNodePolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy"
  role       = aws_iam_role.ms-node.name
}

resource "aws_iam_role_policy_attachment" "ms-node-AmazonEKS_CNI_Policy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy"
  role       = aws_iam_role.ms-node.name
}

[...]
resource "aws_iam_role_policy_attachment" "ms-node-ContainerRegistryReadOnly" {
[...]
  policy_arn = "arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly"
  role       = aws_iam_role.ms-node.name
}
```

The role and policies in this Terraform snippet will allow any nodes that are created to communicate with Amazon's container registries and VM services. We need these policies because the nodes in our Kubernetes system will need to be able to provision computing resources and access containers in order to run services. For more details on the IAM role for EKS worker nodes, check out the AWS EKS documentation.

Now that we have our node's role and policy resources defined, we can write the declaration for a node group that uses them. In EKS, a managed node group needs to specify the types of compute and storage resources it will use along with some defined limits for the number of individual nodes or VMs that can be created

automatically. This is important because we are letting EKS automatically provision and scale our nodes. We don't want to inadvertently consume massive amounts of AWS resources and end up with a correspondingly massive bill.

We could hardcode all of these parameters in our module, but instead we'll use input variables as values for the size limits, disk size, and CPU types. That way we'll be able to use the same Kubernetes module to create different kinds of environments. For example, a development environment can be set up to use minimal resources, while a production environment can be more robust.

Add the Terraform code in Example 7-15 to the end of the module's *main.tf* file to define our EKS node group.

Example 7-15. module-aws-kubernetes/main.tf (node group)

```
resource "aws_eks_node_group" "ms-node-group" {
  cluster_name    = aws_eks_cluster.ms-up-running.name
  node_group_name = "microservices"
  node_role_arn   = aws_iam_role.ms-node.arn
  subnet_ids      = var.nodegroup_subnet_ids

  scaling_config {
    desired_size = var.nodegroup_desired_size
    max_size     = var.nodegroup_max_size
    min_size     = var.nodegroup_min_size
  }

  disk_size      = var.nodegroup_disk_size
  instance_types = var.nodegroup_instance_types

  depends_on = [
    aws_iam_role_policy_attachment.ms-node-AmazonEKSWorkerNodePolicy,
    aws_iam_role_policy_attachment.ms-node-AmazonEKS_CNI_Policy,
    aws_iam_role_policy_attachment.ms-node-AmazonEC2ContainerRegistryReadOnly,
  ]
}
```

The node group declaration is the last part of our EKS configuration. We have enough here to be able to call this module from our sandbox environment and instantiate a running Kubernetes cluster on the AWS EKS service. Our module's outputs will return the values that are needed to connect to the node group once it's running. But it's also useful to provide those connection details in a configuration file for the `kubectl` CLI that most operators use for Kubernetes management.

Our last step is to generate a *kubeconfig* file that we'll be able to use to connect to the cluster. Append the code in Example 7-16 to your module's *main.tf* file.

Example 7-16. module-aws-kubernetes/main.tf (generate kubeconfig)

```
# Create a kubeconfig file based on the cluster that has been created
resource "local_file" "kubeconfig" {
  content  = <<KUBECONFIG_END
apiVersion: v1
clusters:
- cluster:
    "certificate-authority-data: >
   ${aws_eks_cluster.ms-up-running.certificate_authority.0.data}"
    server: ${aws_eks_cluster.ms-up-running.endpoint}
  name: ${aws_eks_cluster.ms-up-running.arn}
contexts:
- context:
    cluster: ${aws_eks_cluster.ms-up-running.arn}
    user: ${aws_eks_cluster.ms-up-running.arn}
  name: ${aws_eks_cluster.ms-up-running.arn}
current-context: ${aws_eks_cluster.ms-up-running.arn}
kind: Config
preferences: {}
users:
- name: ${aws_eks_cluster.ms-up-running.arn}
  user:
    exec:
      apiVersion: client.authentication.k8s.io/v1alpha1
      command: aws-iam-authenticator
      args:
        - "token"
        - "-i"
        - "${aws_eks_cluster.ms-up-running.name}"
    KUBECONFIG_END
  filename = "kubeconfig"
}
```

This code looks complicated, but it's actually fairly simple. We are using a special Terraform resource called `local_file` to create a file named *kubeconfig*. We are then populating *kubeconfig* with YAML content that defines the connection parameters for our Kubernetes cluster. Notice that we are getting the values for the YAML file from the EKS resources that we created in the module.

When Terraform runs this block of code, it will create a *kubeconfig* file in a local directory. We'll be able to use that file to connect to the Kubernetes environment from CLI tools. We made a special provision for this file when we built our pipeline in Chapter 6. When you run the infrastructure pipeline, you'll be able to download this populated configuration file and use it to connect to the cluster. This configuration file will make it a lot easier for you to connect to the cluster from your machine.

We're almost done writing our Kubernetes service module; all that's left is to define the variables.

Kubernetes module variables

To declare the variables for our Kubernetes module, create a file called *variables.tf* in your `module-aws-kubernetes` repository and add the code in Example 7-17.

Example 7-17. module-aws-kubernetes/variables.tf

```
variable "aws_region" {
  type        = string
  default     = "eu-west-2"
}

variable "env_name" {
  type = string
}

variable "cluster_name" {
  type = string
}

variable "ms_namespace" {
  type    = string
  default = "microservices"
}

variable "vpc_id" {
  type = string
}

variable "cluster_subnet_ids" {
  type = list(string)
}

variable "nodegroup_subnet_ids" {
  type = list(string)
}

variable "nodegroup_desired_size" {
  type    = number
  default = 1
}

variable "nodegroup_min_size" {
  type    = number
  default = 1
}

variable "nodegroup_max_size" {
  type    = number
  default = 5
}
```

```
variable "nodegroup_disk_size" {
  type = string
}

variable "nodegroup_instance_types" {
  type = list(string)
}
```

Our AWS Kubernetes module is now fully written. As we did for our network module, we'll take a moment to clean up the formatting and validate the syntax of the code by running the following Terraform commands:

```
module-aws-kubernetes$ terraform fmt
[...]
module-aws-kubernetes$ terraform init
[...]
module-aws-kubernetes$ terraform validate
Success! The configuration is valid.
```

When you are satisfied that the code is valid, commit your changes and push them to GitHub, so that we can use this module in the sandbox environment:

```
$ git add .
$ git commit -m "kubernetes module complete"
$ git push origin
```

With the EKS module ready to go, we can go back to our sandbox Terraform file and use it.

Create a sandbox Kubernetes cluster

Now that our complex Kubernetes system is wrapped up in a simple module, the work of setting it up in our sandbox environment is pretty simple. All we'll need to do is call our module with the input parameters that we want. Remember that our sandbox environment is defined in its own code repository and has its own Terraform file called *main.tf* which we've used to set up the network. We'll be editing that file again, but this time we'll add a call to the Terraform module.

If you recall, we gave some of our input variables default values. To keep things simple, we'll just use those default values in our sandbox environment. We'll also need to pass some of the output variables from our network module into this Kubernetes module so that it installs the cluster on the network we've just created. But beyond those inputs, you'll need to define the `aws_region` value for your installation. This should be the same as the value you used for the network module and the backend configuration. You'll also need to set the source parameter to point to your GitHub-hosted module.

Update the *main.tf* file of your sandbox environment so that it uses the Kubernetes module you've just created. You can add the module reference immediately after the

`#EKS Configuration` placeholder we put in the file earlier. You'll also need to replace the token **`{YOUR_EKS_MODULE_PATH}`** with the path to your module's GitHub repository (see Example 7-18).

Example 7-18. env-sandbox/main.tf (Kubernetes)

```
...

# Network Configuration
...

# EKS Configuration
module "aws-eks" {
  source = "*github.com/{YOUR_EKS_MODULE_PATH}*"

  ms_namespace       = "microservices"
  env_name           = local.env_name
  aws_region         = local.aws_region
  cluster_name       = local.k8s_cluster_name
  vpc_id             = module.aws-network.vpc_id
  cluster_subnet_ids = module.aws-network.subnet_ids

  nodegroup_subnet_ids     = module.aws-network.private_subnet_ids
  nodegroup_disk_size      = "20"
  nodegroup_instance_types = ["t3.medium"]
  nodegroup_desired_size   = 1
  nodegroup_min_size       = 1
  nodegroup_max_size       = 3
}

# GitOps Configuration
```

Now you can commit and push this file into your CI/CD infrastructure pipeline and create a working EKS cluster. Don't forget that you'll need to use a tag to get the build to kick off. For example, you can run the following commands to create a 1.1 version of the infrastructure:

```
$ git add .
$ git commit -m "initial k8s release"
$ git push
$ git tag -a v1.1 -m "k8s build"
$ git push origin v1.1
```

Be prepared to wait for a few minutes for a result as provisioning a new EKS cluster can take up to 10 to 15 minutes. When it's done, you'll have a powerful container-based infrastructure up and running, ready to run your microservices resiliently.

The AWS EKS cluster we've defined here will accrue charges even when it's idle. We recommend that you destroy the environment when you are not using it. You'll find instructions for doing that in "Cleaning Up the Infrastructure" on page 79.

You can test that the cluster has been provisioned by running the following AWS CLI command:

```
$  aws eks list-clusters
```

If all has gone well, you'll get the following response:

```
{
    "clusters": [
        "ms-cluster-sandbox"
    ]
}
```

Our final step is to install a GitOps deployment tool that will come in handy when it's time to release our services into our environment's Kubernetes cluster.

Setting Up Argo CD

As we mentioned earlier, we're going to complete our infrastructure setup with a GitOps server that we'll use later in the book. We'll continue to follow the module pattern by creating a Terraform module for Argo CD that we can call to bootstrap the server in our sandbox environment. Unlike the other modules, we'll be installing Argo CD on the Kubernetes system that we've just instantiated.

To do that, we'll need to let Terraform know that we're using a different host. Up until now, we've been using the AWS provider, which lets Terraform communicate with AWS through its API. For our Argo CD installation we'll use a Kubernetes provider; this enables Terraform to issue Kubernetes commands and install the application to our new cluster. We'll also use a package-management system called Helm to do the installation. We'll introduce Helm a little bit later, but for now, we'll need to set up Terraform to use it as a provider.

We'll install this resource into the Kubernetes cluster rather than on the AWS platform.

That means we won't be using the AWS provider. Instead, we'll use Terraform's Kubernetes and Helm providers.

A completed version of this module is available in this book's GitHub repository.

Create a file called *main.tf* file in the root directory of the `module-argo-cd` Git repository that you created earlier. Add the code in Example 7-19 to set up the providers we need for the installation.

Example 7-19. module-argo-cd/main.tf

```
provider "kubernetes" {
  load_config_file       = false
  cluster_ca_certificate = base64decode(var.kubernetes_cluster_cert_data)
  host                   = var.kubernetes_cluster_endpoint
  exec {
    api_version = "client.authentication.k8s.io/v1alpha1"
    command     = "aws-iam-authenticator"
    args        = ["token", "-i", "${var.kubernetes_cluster_name}"]
  }
}

provider "helm" {
  kubernetes {
    load_config_file       = false
    cluster_ca_certificate = base64decode(var.kubernetes_cluster_cert_data)
    host                   = var.kubernetes_cluster_endpoint
    exec {
      api_version = "client.authentication.k8s.io/v1alpha1"
      command     = "aws-iam-authenticator"
      args        = ["token", "-i", "${var.kubernetes_cluster_name}"]
    }
  }
}
```

To configure the Kubernetes provider, we're using the properties of the EKS cluster that we provisioned earlier. These properties let Terraform know it needs to use the AWS authenticator to connect to the cluster along with the certificate that we've provided.

As we mentioned earlier, we're also using a provider for Helm. Helm is a popular way of describing a Kubernetes deployment and for distributing Kubernetes applications as packages. It's similar to other package-management tools, such as `apt-get` in the Linux world, and is designed to make installation of Kubernetes-based applications simple and easy. To configure our Helm provider, we simply need to provide a few Kubernetes connection parameters.

A Helm deployment is called a *chart*. We'll be using a Helm chart provided by the Argo CD community to install the Argo CD server. Add the code in Example 7-20 to the *main.tf* file to complete the installation declaration.

Example 7-20. module-argo-cd/main.tf (Helm)

```
resource "kubernetes_namespace" "example" {
  metadata {
    name = "argo"
  }
}

resource "helm_release" "argocd" {
  name       = "msur"
  chart      = "argo-cd"
  repository = "https://argoproj.github.io/argo-helm"
  namespace  = "argo"
}
```

This code creates a namespace for the Argo CD installation and uses the Helm provider to perform the installation. All that's left to complete the Argo CD module is to define some variables.

Variables for Argo CD

Create a file called *variables.tf* in your Argo CD module repository and add the code in Example 7-21.

Example 7-21. module-argo-cd/variables.tf

```
variable "kubernetes_cluster_id" {
  type = string
}

variable "kubernetes_cluster_cert_data" {
  type = string
}

variable "kubernetes_cluster_endpoint" {
  type = string
}

variable "kubernetes_cluster_name" {
  type = string
}

variable "eks_nodegroup_id" {
  type = string
}
```

We need to define these variables so that we can configure the Kubernetes and Helm providers in our code. So we'll need to grab them from the Kubernetes module's output when we call it in our sandbox's Terraform file. Before we get to that step, let's

format and validate the code we've written in the same way as we did for our other modules:

```
module-argocd$ terraform fmt
[...]
module-argocd$ terraform init
[...]
module-argocd$ terraform validate
Success! The configuration is valid.
```

When you are satisfied that the code is valid, commit your code changes and push them to the GitHub repository so that we can use the module in our sandbox environment:

```
$ git add .
$ git commit -m "ArgoCD module init"
$ git push origin
```

Now, as we've done before, we just need to call this module from our sandbox definition.

Installing Argo CD in the sandbox

We want the Argo CD installation to happen as part of our sandbox environment bootstrapping, so we need to call the module from the Terraform definition in our sandbox environment. Add the code in Example 7-22 to the end of your sandbox module's *main.tf* file to install Argo CD. Don't forget to use your module's GitHub repository path in the source property of the module definition.

Example 7-22. env-sandbox/main.tf (Argo CD)

```
...

# Network Configuration
...

# EKS Configuration
...

# GitOps Configuration
module "argo-cd-server" {
  source = "*github.com/{YOUR_ARGOCD_MODULE_PATH}*"

  kubernetes_cluster_id        = module.aws-eks.eks_cluster_id
  kubernetes_cluster_name      = module.aws-eks.eks_cluster_name
  kubernetes_cluster_cert_data = module.aws-eks.eks_cluster_certificate_data
  kubernetes_cluster_endpoint  = module.aws-eks.eks_cluster_endpoint
  eks_nodegroup_id = module.aws-eks.eks_cluster_nodegroup_id
}
```

Now, you can tag, commit, and push the Terraform file into your CI/CD pipeline just like you've done before. For example, the following command will push a v1.2 tag into the repository and kick off the pipeline process:

You'll need to wait for the EKS build to complete before tagging and committing these Argo CD sandbox changes. Otherwise, there won't be a Kubernetes cluster for Argo CD to be deployed to.

```
$ git add .
$ git commit -m "initial ArgoCD release"
$ git push origin
$ git tag -a v1.2 -m "ArgoCD build"
$ git push origin v1.2
```

When our pipeline is finished applying changes, you'll have a GitOps server that will help deploy microservices easier and more reliably. With that step completed, we've finished defining and provisioning the sandbox environment. All that's left is to test it and see if it works.

Testing the Environment

Before we finish with our infrastructure implementation, it's a good idea to run a test and make sure that the environment has been provisioned as expected. We'll do this by verifying that we can log in to the Argo CD web console. That will prove that the entire stack is running and operational. But in order to do that, we'll need to set up our `kubectl` CLI application.

Earlier in this chapter, when we were creating the Terraform code for our Kubernetes module, we added a local file resource to create a *kubeconfig* file. Now, we need to download that file so that we can connect to the EKS cluster using the `kubectl` application.

To retrieve this file, navigate to your sandbox GitHub repository in your browser and click on the Actions tab. You should see a list of builds with your latest run at the top of the screen. When you select the build that you just performed, you should see an artifact called "kubeconfig" that you can click and download.

If you're having trouble finding the page to download the artifact, try following the instructions in the GitHub documentation.

GitHub will package the artifact as a ZIP file, so after downloading it you'll need to decompress the package. Inside the ZIP file you should find a file called *kubeconfig*. To use it, you just need to set an environment variable called `KUBECONFIG` that points to it. This will let the Kubernetes command-line application know where to find it. For example, if the *kubeconfig* file is in your *~/Downloads* directory, use the following value:

```
$ export KUBECONFIG=~/Downloads/kubeconfig
```

If you like, you can copy the *kubeconfig* file to *~/.kube/config* and avoid having to set an environment variable. Just make sure you aren't overwriting a Kubernetes configuration you're already using.

You can test that everything runs as expected by issuing the following command:

```
$ kubectl get svc
```

You should see something like the following in response:

```
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   172.20.0.1   <none>        443/TCP   2h
```

This shows us that our network and EKS services were provisioned and we were able to successfully connect to the cluster. To get this information, `kubectl` makes an API call to the Kubernetes cluster we've just created. Getting this response back is proof that our cluster is up and running. As a final test, we'll check to make sure that Argo CD has been installed in the cluster. Run the following command to verify that the Argo CD pods are running:

```
$ kubectl get pods -n "argo"
NAME                                                READY STATUS    RESTARTS
msur-argocd-application-controller-5bddfb78fc-9jpzj 1/1   Running            0
msur-argocd-dex-server-84cd5fc9b9-bjzrm             1/1   Running            0
msur-argocd-redis-dc867dd9c-rpgww                   1/1   Running            0
msur-argocd-repo-server-75474975cc-j7lws            1/1   Running            0
msur-argocd-server-5cc998b478-wvkrr                 1/1   Running            0
```

A Kubernetes Pod represents a deployable unit, consisting of one or more container images.

Later in the process, we'll get a chance to use Argo CD, the Kubernetes cluster, and the rest of the infrastructure we've designed. But now that we know our pipeline and configurations work, it's time to tear it all down. Don't worry, though: with our code written, it will be easy to create our environment again when we need it.

Cleaning Up the Infrastructure

We now have our infrastructure up and running. But, if you aren't planning on using it right away, it's a good idea to clean things up so you don't incur any costs to have it running. In particular, the elastic IP addresses that we used for our network can be costly if we leave them up. Since our environment is now completely defined in Terraform declarative files, we can re-create it in the same way whenever we need it, so destroying the existing environment is a low-risk activity.

Terraform will automatically destroy resources in the correct order for us because it has internally created a dependency graph. To destroy the sandbox environment, use the following steps:

1. Navigate to the working directory of your sandbox environment code on your machine. This is the same directory you used in "Installing Argo CD in the sandbox" on page 76.
2. Pull the latest version of the code from the repository:

   ```
   env-sandbox$ git pull
   ```

   Install the Terraform providers that our environment code uses (we'll need these so we can destroy the resources):

   ```
   env-sandbox$ terraform init
   ```
3. After Terraform has finished downloading plug-ins, enter the following command to destroy the sandbox environment:

   ```
   env-sandbox$ terraform destroy
   ```
4. Terraform will display the resources that it will destroy. You'll need to say `yes` to continue to the removal process. It will probably take about five minutes to complete. When it's done all of the AWS resources that we created will be gone.

We're able to destroy these AWS resources from our local machine because we have AWS access and secret keys stored in a local credentials file. This shouldn't be the case for a production or secured environment.

5. When it's done, you'll see a message that looks like this:

   ```
   Destroy complete! Resources: 29 destroyed.
   ```
6. To verify that the EKS resources have been removed, you can run the following AWS CLI command to list EKS clusters:

   ```
   $ aws eks list-clusters
   ```

You should get back a response indicating that there are no EKS clusters left in your instance:

```
{
    "clusters": []
}
```

You can also run the following commands to double-check that the other billable resources have been removed:

```
$ aws ec2 describe-vpcs --filters Name=cidr,Values=10.10.0.0/16
$ aws elbv2 describe-load-balancers
```

It's not absolutely necessary to run these CLI commands if `terraform destroy` returns successfully. We have included them so you can double-check that they are really gone, so you will not be billed unexpectedly.

If something has gone wrong, you'll need to use the AWS console and remove the resources manually. Consult the AWS documentation if you have trouble deleting resources through the console.

Summary

We did a lot in this chapter. We created a Terraform module for our software-defined network that spanned two availability zones in a single region. Next, we created a module that instantiates an AWS EKS cluster for Kubernetes. We also implemented an Argo CD GitOps server into the cluster using a Helm package. Finally, we implemented a sandbox environment as code that uses all of these modules in a declarative, immutable way.

We went into a lot of detail with the Terraform code in this chapter. We did that so you could get a feel for what it takes to define an environment using infrastructure as code, immutability, and a CI/CD pipeline. We also wanted you to get hands on with the Terraform module pattern and some of the design decisions you'll need to make for your infrastructure. As we learn more about the microservices we are deploying, we may need additional infrastructure modules, but later in the book we'll use prewritten, hosted code instead of walking through it all line by line.

In Chapter 8, we'll get back to our example microservices and start the work of developing them. When we're done, we'll be able to release them into the infrastructure we've just designed.

About the Authors

Ronnie Mitra is an author, strategist, and consultant with over 25 years of experience working with web and connectivity technologies. He is the coauthor of *Microservice Architecture* and *Continuous API Management* (both O'Reilly).

Irakli Nadareishvili is the vice president of Core Innovation at Capital One Financial Corporation, leading the teams responsible for building Capital One's modern, cloud native, microservices-based core banking platform. Before Capital One, Irakli was cofounder and CTO of ReferWell, a successful New York City–based health technology startup, and held technology leadership roles at CA Technologies and NPR. Irakli is the coauthor of *Microservice Architecture* (O'Reilly). You can follow Irakli on Twitter at @inadarei.

Colophon

The animal on the cover of *Microservices: Up and Running* is the sparkling violetear hummingbird (*Colibri coruscans*). This hummingbird lives in a range that runs along the northwestern coast of South America, in higher-elevation habitats among the Andes mountains. Known in the Quechua language as *Siwar q'inti*, these hummingbirds have a place in local folklore as a sign of good luck.

Sparkling violetears are iridescent green with purple markings on the head and chest. The longer purple feathers at their ears extend outward from their heads during display. Large for hummingbirds, they average about five to six inches long, and weigh about a quarter ounce. Females lay two eggs in a nest of their own making, and incubate the eggs. The chicks fledge from the nest at three weeks.

Because they live at higher, colder altitudes, sparkling violetears are among the species of hummingbirds that enter a deep torpor each night to sleep. In this hibernation-like state of reduced body functions and a near-acclimation to surrounding cold temperatures, which it then reverses at dawn, the bird is able to survive long, cold nights without the food it would otherwise need to stay warm. The mechanisms by which they accomplish this complicated feat are the subject of ongoing scientific studies.

The sparkling violetear is common across its range, and is rated by the IUCN to be of Least Concern. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The color illustration on the cover is by Karen Montgomery, based on a black-and-white engraving from *Wood's Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.