O'REILLY®

Compliments of NGINX

Production-Ready Microservices

BUILDING STANDARDIZED SYSTEMS ACROSS AN ENGINEERING ORGANIZATION

FREE CHAPTERS

Susan J. Fowler

Production-Ready Microservices

Building Standardized Systems Across an Engineering Organization

This Excerpt contains Chapters 1, 3, 4, 7, and Appendix A of the book *Production-Ready Microservices*. The complete book is available at *oreilly.com* and through other retailers.

Susan J. Fowler

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

Production-Ready Microservices
by Susan J. Fowler

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/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Nan Barber and Brian Foster
Production Editor: Kristen Brown
Copyeditor: Amanda Kersey
Proofreader: Jasmine Kwityn
Indexer: Wendy Catalano
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

December 2016: First Edition

Revision History for the First Edition
2016-11-23: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491965979* for release details.

978-1-491-96597-9

[LSI]

Table of Contents

CHAPTER 1
Microservices

In the past few years, the technology industry has witnessed a rapid change in applied, practical distributed systems architecture that has led industry giants (such as Netflix, Twitter, Amazon, eBay, and Uber) away from building monolithic applications to adopting microservice architecture. While the fundamental concepts behind microservices are not new, the contemporary application of microservice architecture truly is, and its adoption has been driven in part by scalability challenges, lack of efficiency, slow developer velocity, and the difficulties with adopting new technologies that arise when complex software systems are contained within and deployed as one large monolithic application.

Adopting microservice architecture, whether from the ground up or by splitting an existing monolithic application into independently developed and deployed microservices, solves these problems. With microservice architecture, an application can easily be scaled both horizontally and vertically, developer productivity and velocity increase dramatically, and old technologies can easily be swapped out for the newest ones.

As we will see in this chapter, the adoption of microservice architecture can be seen as a natural step in the scaling of an application. The splitting of a monolithic application into microservices is driven by scalability and efficiency concerns, but microservices introduce challenges of their own. A successful, scalable microservice ecosystem requires that a stable and sophisticated infrastructure be in place. In addition, the organizational structure of a company adopting microservices must be radically changed to support microservice architecture, and the team structures that spring from this can lead to siloing and sprawl. The largest challenges that microservice architecture brings, however, are the need for standardization of the architecture of the services themselves, along with requirements for each microservice in order to ensure trust and availability.

From Monoliths to Microservices

Almost every software application written today can be broken into three distinct elements: a *frontend* (or *client-side*) piece, a *backend* piece, and some type of *datastore* (Figure 1-1). Requests are made to the application through the client-side piece, the backend code does all the heavy lifting, and any relevant data that needs to be stored or accessed (whether temporarily in memory of permanently in a database) is sent to or retrieved from wherever the data is stored. We'll call this *the three-tier architecture*.

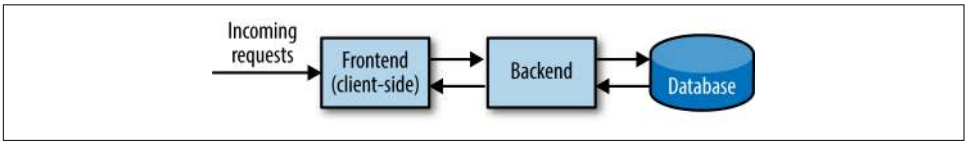

Figure 1-1. Three-tier architecture

There are three different ways these elements can be combined to make an application. Most applications put the first two pieces into one codebase (or repository), where all client-side and backend code are stored and run as one executable file, with a separate database. Others separate out all frontend, client-side code from the backend code and store them as separate logical executables, accompanied by an external database. Applications that don't require an external database and store all data in memory tend to combine all three elements into one repository. Regardless of the way these elements are divided or combined, the *application* itself is considered to be the sum of these three distinct elements.

Applications are usually architected, built, and run this way from the beginning of their lifecycles, and the architecture of the application is typically independent of the product offered by the company or the purpose of the application itself. These three architectural elements that comprise the three-tier architecture are present in every website, every phone application, every backend and frontend and strange enormous enterprise application, and are found as one of the permutations described.

In the early stages, when a company is young, its application(s) simple, and the number of developers contributing to the codebase is small, developers typically share the burden of contributing to and maintaining the codebase. As the company grows, more developers are hired, new features are added to the application, and three significant things happen.

First comes an increase in the operational workload. Operational work is, generally speaking, the work associated with running and maintaining the application. This usually leads to the hiring of operational engineers (system administrators, TechOps engineers, and so-called "DevOps" engineers) who take over the majority of the operational tasks, like those related to hardware, monitoring, and on call.

The second thing that happens is a result of simple mathematics: adding new features to your application increases both the number of lines of code in your application and the complexity of the application itself.

Third is the necessary horizontal and/or vertical scaling of the application. Increases in traffic place significant scalability and performance demands on the application, requiring that more servers host the application. More servers are added, a copy of the application is deployed to each server, and load balancers are put into place so that the requests are distributed appropriately among the servers hosting the application (see Figure 1-2, containing a frontend piece, which may contain its own load-balancing layer, a backend load-balancing layer, and the backend servers). Vertical scaling becomes a necessity as the application begins processing a larger number of tasks related to its diverse set of features, so the application is deployed to larger, more powerful servers that can handle CPU and memory demands (Figure 1-3).

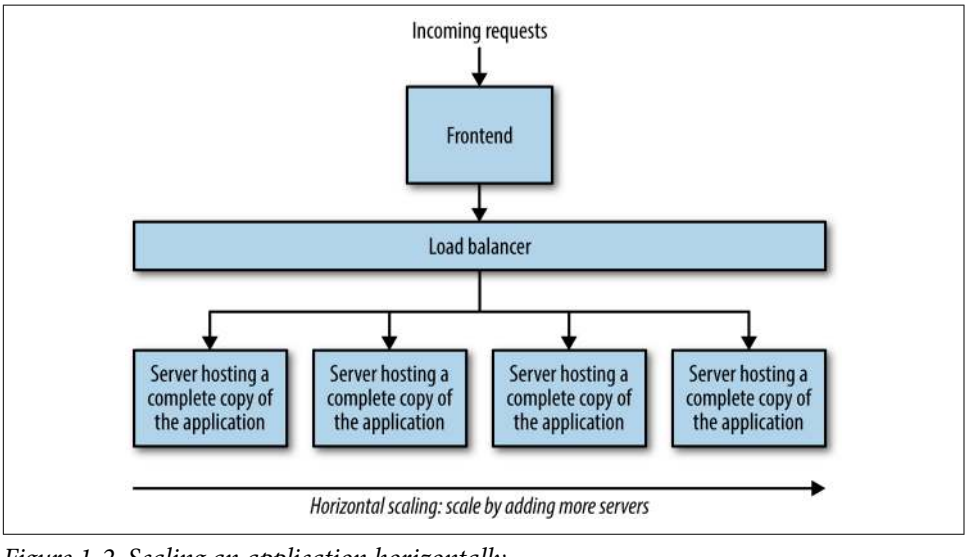

Figure 1-2. Scaling an application horizontally

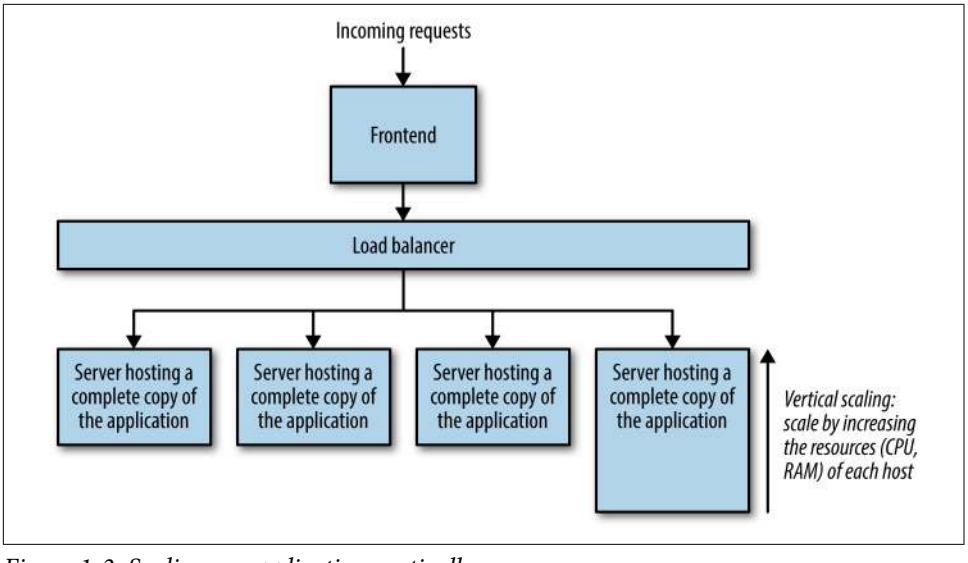

Figure 1-3. Scaling an application vertically

As the company grows, and the number of engineers is no longer in the single, double, or even triple digits, things start to get a little more complicated. Thanks to all the features, patches, and fixes added to the codebase by the developers, the application is now thousands upon thousands of lines long. The complexity of the application is growing steadily, and hundreds (if not thousands) of tests must be written in order to ensure that any change made (even a change of one or two lines) doesn't compromise the integrity of the existing thousands upon thousands of lines of code. Development and deployment become a nightmare, testing becomes a burden and a blocker to the deployment of even the most crucial fixes, and technical debt piles up quickly. Applications whose lifecycles fit into this pattern (for better or for worse) are fondly (and appropriately) referred to in the software community as *monoliths*.

Of course, not all monolithic applications are bad, and not every monolithic application suffers from the problems listed, but monoliths that don't hit these issues at some point in their lifecycle are (in my experience) pretty rare. The reason most monoliths are susceptible to these problems is because the nature of a monolith is directly opposed to *scalability* in the most general possible sense. Scalability requires *concurrency* and *partitioning*: the two things that are difficult to accomplish with a monolith.

Scaling an Application

Let's break this down a bit.

The goal of any software application is to process tasks of some sort. Regardless of what those tasks are, we can make a general assumption about how we want our application to handle them: it needs to process them efficiently.

To process tasks efficiently, our application needs to have some kind of *concurrency*. This means that we can't have just one process that does all the work, because then that process will pick up one task at a time, complete all the necessary pieces of it (or fail!), and then move onto the next—this isn't efficient at all! To make our application efficient, we can introduce concurrency so that each task can be broken up into smaller pieces.

The second thing we can do to process tasks efficiently is to divide and conquer by introducing *partitioning*, where each task is not only broken up into small pieces but can be processed in parallel. If we have a bunch of tasks, we can process them all at the same time by sending them to a set of workers that can process them in parallel. If we need to process more tasks, we can easily scale with the demand by adding additional workers to process the new tasks without affecting the efficiency of our system.

Concurrency and partitioning are difficult to support when you have one large application that needs to be deployed to every server, which needs to process any type of task. If your application is even the slightest bit complicated, the only way you can scale it with a growing list of features and increasing traffic is to scale up the hardware that the application is deployed to.

To be truly efficient, the best way to scale an application is to break it into many small, independent applications that each do one type of task. Need to add another step to the overall process? Easy enough: just make a new application that only does that step! Need to handle more traffic? Simple: add more workers to each application!

Concurrency and partitioning are difficult to support in a monolithic application, which prevents monolithic application architecture from being as efficient as we need it to be.

We've seen this pattern emerge at companies like Amazon, Twitter, Netflix, eBay, and Uber: companies that run applications across not hundreds, but thousands, even hundreds of thousands of servers and whose applications have evolved into monoliths and hit scalability challenges. The challenges they faced were remedied by abandoning monolithic application architecture in favor of *microservices*.

The basic concept of a microservice is simple: it's a small application that does one thing only, and does that one thing well. A microservice is a small component that is

easily replaceable, independently developed, and independently deployable. A microservice cannot live alone, however—no microservice is an island—and it is part of a larger system, running and working alongside other microservices to accomplish what would normally be handled by one large standalone application.

The goal of microservice architecture is to build a set of small applications that are each responsible for performing one function (as opposed to the traditional way of building one application that does everything), and to let each microservice be autonomous, independent, and self-contained. The core difference between a monolithic application and microservices is this: a monolithic application (Figure 1-4) will contain all features and functions within one application and one codebase, all deployed at the same time, with each server hosting a complete copy of the entire application, while a microservice (Figure 1-5) contains only one function or feature and lives in a *microservice ecosystem* along with other microservices that each perform one function or feature.

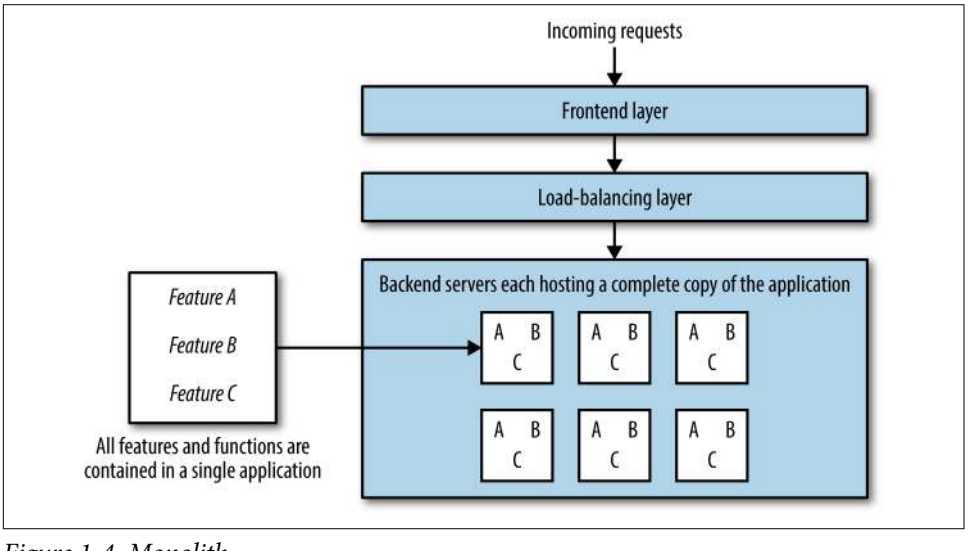

Figure 1-4. Monolith

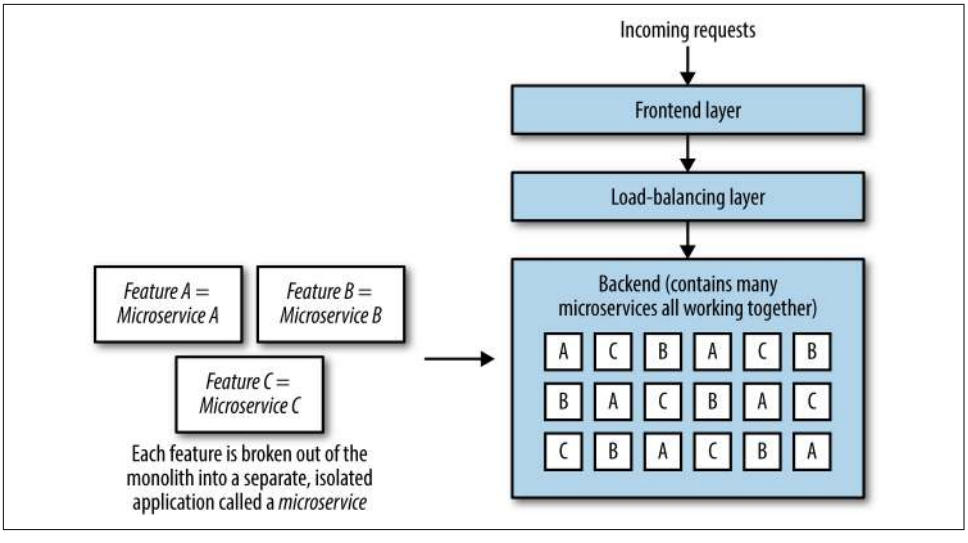

Figure 1-5. Microservices

There are numerous benefits to adopting microservice architecture—including (but not limited to) reduced technical debt, improved developer productivity and velocity, better testing efficiency, increased scalability, and ease of deployment—and companies that adopt microservice architecture usually do so after having built one application and hitting scalability and organizational challenges. They begin with a monolithic application and then *split the monolith* into microservices.

The difficulties of splitting a monolith into microservices depend entirely on the complexity of the monolithic application. A monolithic application with many features will take a great deal of architectural effort and careful deliberation to successfully break up into microservices, and additional complexity is introduced by the need to reorganize and restructure teams. The decision to move to microservices must always become a company-wide effort.

There are several steps in breaking apart a monolith. The first is to identify the components that should be written as independent services. This is perhaps the most difficult step in the entire process, because while there may be a number of right ways to split the monolith into component services, there are far more wrong ways. The rule of thumb in identifying components is to pinpoint key overall functionalities of the monolith, then split those functionalities into small independent components. Microservices must be as simple as possible or else the company will risk the possibility of replacing one monolith with several smaller monoliths, which will all suffer the same problems as the company grows.

Once the key functions have been identified and properly componentized into independent microservices, the organizational structure of the company must be restruc-

tured so that each microservice is staffed by an engineering team. There are several ways to do this. The first method of company reorganization around microservice adoption is to dedicate one team to each microservice. The size of the team will be determined completely by the complexity and workload of the microservice and should be staffed by enough developers and site reliability engineers so that both feature development and the on-call rotation of the service can be managed without burdening the team. The second is to assign several services to one team and have that team develop the services in parallel. This works best when the teams are organized around specific products or business domains, and are responsible for developing any services related to those products or domains. If a company chooses the second method of reorganization, it needs to make sure that developers aren't overworked and don't face task, outage, or operational fatigue.

Another important part of microservice adoption is the creation of a *microservice ecosystem*. Typically (or, at least, hopefully), a company running a large monolithic application will have a dedicated infrastructure organization that is responsible for designing, building, and maintaining the infrastructure that the application runs on. When a monolith is split into microservices, the responsibilities of the infrastructure organization for providing a stable platform for microservices to be developed and run on grows drastically in importance. The infrastructure teams must provide microservice teams with stable infrastructure that abstracts away the majority of the complexity of the interactions between microservices.

Once these three steps have been completed—the componentization of the application, the restructuring of engineering teams to staff each microservice, and the development of the infrastructure organization within the company—the migration can begin. Some teams choose to pull the relevant code for their microservice directly from the monolith and into a separate service, and shadow the monolith's traffic until they are convinced that the microservice can perform the desired functionality on its own. Other teams choose to build the service from scratch, starting with a clean slate, and shadow traffic or redirect after the service has passed appropriate tests. The best approach to migration depends on the functionality of the microservice, and I have seen both approaches work equally well in most cases, but the real key to a successful migration is thorough, careful, painstakingly documented planning and execution, along with the realization that a complete migration of a large monolith can take several long years.

With all the work involved in splitting a monolith into microservices, it may seem better to begin with microservice architecture, skip all of the painful scalability challenges, and avoid the microservice migration drama. This approach may turn out all right for some companies, but I want to offer several words of caution. Small companies often do not have the necessary infrastructure in place to sustain microservices, even at a very small scale: good microservice architecture requires stable, often very complex, infrastructure. Such stable infrastructure requires a large, dedicated team

whose cost can typically be sustained only by companies that have reached the scalability challenges that justify the move to microservice architecture. Small companies simply will not have enough operational capacity to maintain a microservice ecosystem. Furthermore, it's extraordinarily difficult to identify key areas and components to build into microservices when a company is in the early stages: applications at new companies will not have many features, nor many separate areas of functionality that can be split appropriately into microservices.

Microservice Architecture

The *architecture of a microservice* (Figure 1-6) is not very different from the standard application architecture covered in the first section of this chapter (Figure 1-1). Each and every microservice will have three components: a frontend (client-side) piece, some backend code that does the heavy lifting, and a way to store or retrieve any relevant data.

The frontend, client-side piece of a microservice is not your typical frontend application, but rather an *application programming interface* (API) with static *endpoints*. Well-designed microservice APIs allow microservices to easily and effectively interact, sending requests to the relevant API endpoint(s). For example, a microservice that is responsible for customer data might have a *get_customer_information* endpoint that other services could send requests to in order to retrieve information about customers, an *update_customer_information* endpoint that other services could send requests to in order to update the information for a specific customer, and a *delete_customer_information* endpoint that services could use to delete a customer's information.

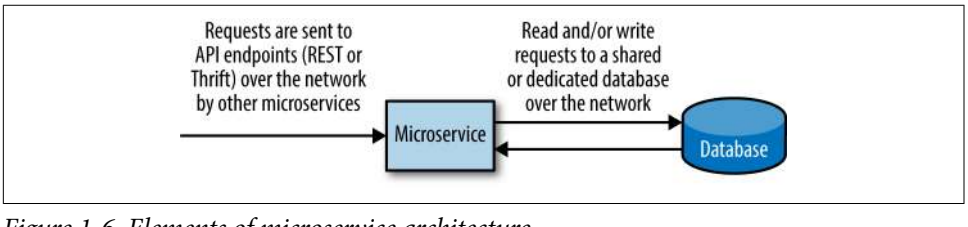

Figure 1-6. Elements of microservice architecture

These endpoints are separated out in architecture and theory alone, not in practice, for they live alongside and as part of all the backend code that processes every request. For our example microservice that is responsible for customer data, a request sent to the *get_customer_information* endpoint would trigger a task that would process the incoming request, determine any specific filters or options that were applied in the request, retrieve the information from a database, format the information, and return it to the client (microservice) that requested it.

Most microservices will store some type of data, whether in memory (perhaps using a cache) or an external database. If the relevant data is stored in memory, there's no need to make an extra network call to an external database, and the microservice can easily return any relevant data to a client. If the data is stored in an external database, the microservice will need to make another request to the database, wait for a response, and then continue to process the task.

This architecture is necessary if microservices are to work well together. The microservice architecture paradigm requires that a set of microservices work together to make up what would otherwise exist as one large application, and so there are certain elements of this architecture that need to be standardized across an entire organization if a set of microservices is to interact successfully and efficiently.

The API endpoints of microservices should be standardized across an organization. That is not to say that all microservices should have the same specific endpoints, but that the type of endpoint should be the same. Two very common types of API endpoints for microservices are REST or Apache Thrift, and I've seen some microservices that have both types of endpoints (though this is rare, makes monitoring rather complicated, and I don't particularly recommend it). Choice of endpoint type is reflective of the internal workings of the microservice itself, and will also dictate its architecture: it's difficult to build an asynchronous microservice that communicates via HTTP over REST endpoints, for example, which would necessitate adding a messaging-based endpoint to the services as well.

Microservices interact with each other via *remote procedure calls* (RPCs), which are calls over the network designed to look and behave exactly like local procedure calls. The protocols used will be dependent on architectural choices and organizational support, as well as the endpoints used. A microservice with REST endpoints, for example, will likely interact with other microservices via HTTP, while a microservice with Thrift endpoints may communicate with other microservices over HTTP or a more customized, in-house solution.

Avoid Versioning Microservices and Endpoints

A microservice is not a library (it is not loaded into memory at compilation-time or during runtime) but an independent software application. Due to the fast-paced nature of microservice development, versioning microservices can easily become an organizational nightmare, with developers on client services pinning specific (outdated, unmaintained) versions of a microservice in their own code. Microservices should be treated as living, changing things, not static releases or libraries. Versioning of API endpoints is another anti-pattern that should be avoided for the same reasons.

Any type of endpoint and any protocol used to communicate with other microservices will have benefits and trade-offs. The architectural decisions here shouldn't be made by the individual developer who is building a microservice, but should be part of the architectural design of the microservice ecosystem as a whole (we'll get to this in the next section).

Writing a microservice gives the developer a great deal of freedom: aside from any organizational choices regarding API endpoints and communication protocols, developers are free to write the internal workings of their microservice however they wish. It can be written in any language whatsoever—it can be written in Go, in Java, in Erlang, in Haskell—as long as the endpoints and communication protocols are taken care of. Developing a microservice is not all that different from developing a standalone application. There are some caveats to this, as we will see in the final section of this chapter ("Organizational Challenges" on page 20), because developer freedom with regard to language choice comes at a hefty cost to the engineering organization.

In this way, a microservice can be treated by others as a black box: you put some information in by sending a request to one of its endpoints, and you get something out. If you get what you want and need out of the microservice in a reasonable time and without any crazy errors, it has done its job, and there's no need to understand anything further than the endpoints you need to hit and whether or not the service is working properly.

Our discussion of the specifics of microservice architecture will end here—not because this is all there is to microservice architecture, but because each of the following chapters within this book is devoted to bringing microservices to this ideal black-box state.

The Microservice Ecosystem

Microservices do not live in isolation. The environment in which microservices are built, are run, and interact is where they *live*. The complexities of the large-scale microservice environment are on par with the ecological complexities of a rainforest, a desert, or an ocean, and considering this environment as an ecosystem—a *microservice ecosystem*—is beneficial in adopting microservice architecture.

In well-designed, sustainable microservice ecosystems, the microservices are abstracted away from all infrastructure. They are abstracted away from the hardware, abstracted away from the networks, abstracted away from the build and deployment pipeline, abstracted away from service discovery and load balancing. This is all part of the infrastructure of the microservice ecosystem, and building, standardizing, and maintaining this infrastructure in a stable, scalable, fault-tolerant, and reliable way is essential for successful microservice operation.

The infrastructure has to sustain the microservice ecosystem. The goal of all infrastructure engineers and architects must be to remove the low-level operational concerns from microservice development and build a stable infrastructure that can scale, one that developers can easily build and run microservices on top of. Developing a microservice within a stable microservice ecosystem should be just like developing a small standalone application. This requires very sophisticated, top-notch infrastructure.

The microservice ecosystem can be split into four layers (Figure 1-7), though the boundaries of each are not always clearly defined: some elements of the infrastructure will touch every part of the stack. The lower three layers are the infrastructure layers: at the bottom of the stack we find the hardware layer, and on top of that, the communication layer (which bleeds up into the fourth layer), followed by the application platform. The fourth (top) layer is where all individual microservices live.

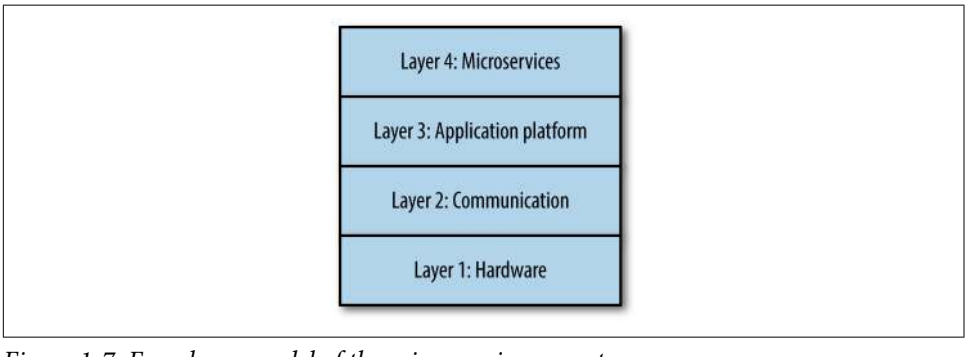

Figure 1-7. Four-layer model of the microservice ecosystem

Layer 1: Hardware

At the very bottom of the microservice ecosystem, we find the *hardware layer*. These are the actual machines, the real, physical computers that all internal tools and all microservices run on. These servers are located on racks within datacenters, being cooled by expensive HVAC systems and powered by electricity. Many different types of servers can live here: some are optimized for databases; others for processing CPU-intensive tasks. These servers can either be owned by the company itself, or "rented" from so-called cloud providers like Amazon Web Services' Elastic Compute Cloud (AWS EC2), Google Cloud Platform (GCP), or Microsoft Azure.

The choice of specific hardware is determined by the owners of the servers. If your company is running your own datacenters, the choice of hardware is your own, and you can optimize the server choice for your specific needs. If you are running servers in the cloud (which is the more common scenario), your choice is limited to whatever hardware is offered by the cloud provider. Choosing between *bare metal* and a *cloud*

provider (or providers) is not an easy decision to make, and cost, availability, reliability, and operational expenses are things that need to be considered.

Managing these servers is part of the hardware layer. Each server needs to have an *operating system* installed, and the operating system should be standardized across all servers. There is no correct, right answer as to which operating system a microservice ecosystem should use: the answer to this question depends entirely on the applications you will be building, the languages they will be written in, and the libraries and tools that your microservices require. The majority of microservice ecosystems run some variant of Linux, commonly CentOS, Debian, or Ubuntu, but a .NET company will, obviously, choose differently. Additional abstractions can be built and layered atop the hardware: resource isolation and resource abstraction (as offered by technologies like Docker and Apache Mesos) also belong in this layer, as do databases (dedicated or shared).

Installing an operating system and *provisioning* the hardware is the first layer on top of the servers themselves. Each host must be provisioned and configured, and after the operating system is installed, a *configuration management* tool (such as Ansible, Chef, or Puppet) should be used to install all of the applications and set all the necessary configurations.

The hosts need proper *host-level monitoring* (using something like Nagios) and *host-level logging* so that if anything happens (disk failure, network failure, or if CPU utilization goes through the roof), problems with the hosts can be easily diagnosed, mitigated, and resolved. Host-level monitoring is covered in greater detail in ???.

Summary of Layer 1: The Hardware Layer

The hardware layer (layer 1) of the microservice ecosystem contains:

- The physical servers (owned by the company or rented from cloud providers)
- Databases (dedicated and/or shared)
- The operating system
- Resource isolation and abstraction
- Configuration management
- Host-level monitoring
- Host-level logging

Layer 2: Communication

The second layer of the microservice ecosystem is the *communication layer*. The communication layer bleeds into all of the other layers of the ecosystem (including the

application platform and microservices layers), because it is where all communication between services is handled; the boundaries between the communication layer and each other layer of the microservice ecosystem are poorly defined. While the boundaries may not be clear, the elements *are* clear: the second layer of a microservice ecosystem always contains the network, DNS, RPCs and API endpoints, service discovery, service registry, and load balancing.

Discussing the network and DNS elements of the communication layer is beyond the scope of this book, so we will be focusing on RPCs, API endpoints, service discovery, service registry, and load balancing in this section.

RPCs, endpoints, and messaging

Microservices interact with one another over the network using *remote procedure calls* (RPCs) or *messaging* to the *API endpoints* of other microservices (or, as we will see in the case of messaging, to a message broker which will route the message appropriately). The basic idea is this: using a specified protocol, a microservice will send some data in a standardized format over the network to another service (perhaps to another microservice's API endpoint) or to a message broker which will make sure that the data is send to another microservice's API endpoint.

There are several microservice communication paradigms. The first is the most common: *HTTP+REST/THRIFT*. In HTTP+REST/THRIFT, services communicate with each other over the network using the *Hypertext Transfer Protocol* (HTTP), and sending requests and receiving responses to or from either specific *representational state transfer* (REST) endpoints (using various methods, like GET, POST, etc.) or specific *Apache Thrift* endpoints (or both). The data is usually formatted and sent as *JSON* (or *protocol buffers*) over HTTP.

HTTP+REST is the most convenient form of microservice communication. There aren't any surprises, it's easy to set up, and is the most stable and reliable—mostly because it's difficult to implement incorrectly. The downside of adopting this paradigm is that it is, by necessity, synchronous (blocking).

The second communication paradigm is *messaging*. Messaging is asynchronous (nonblocking), but it's a bit more complicated. Messaging works the following way: a microservice will send data (a *message*) over the network (HTTP or other) to a *message broker*, which will route the communication to other microservices.

Messaging comes in several flavors, the two most popular being *publish–subscribe* (pubsub) messaging and *request–response* messaging. In pubsub models, clients will *subscribe* to a *topic* and will receive a message whenever a *publisher publishes* a message to that topic. Request–response models are more straightforward, where a client will send a *request* to a service (or message broker), which will *respond* with the information requested. There are some messaging technologies that are a unique blend of both models, like Apache Kafka. Celery and Redis (or Celery with RabbitMQ) can be

used for messaging (and task processing) for microservices written in Python: Celery processes the tasks and/or messages using Redis or RabbitMQ as the broker.

Messaging comes with several significant downsides that must be mitigated. Messaging can be just as scalable (if not more scalable) than HTTP+REST solutions, if it is architected for scalability from the get-go. Inherently, messaging is not as easy to change and update, and its centralized nature (while it may seem like a benefit) can lead to its queues and brokers becoming points of failure for the entire ecosystem. The asynchronous nature of messaging can lead to race conditions and endless loops if not prepared for. If a messaging system is implemented with protections against these problems, it can become as stable and efficient as a synchronous solution.

Service discovery, service registry, and load balancing

In monolithic architecture, traffic only needs to be sent to one application and distributed appropriately to the servers hosting the application. In microservice architecture, traffic needs to be routed appropriately to a large number of different applications, and then distributed appropriately to the servers hosting each specific microservice. In order for this to be done efficiently and effectively, microservice architecture requires three technologies be implemented in the communication layer: *service discovery*, *service registry*, and *load balancing*.

In general, when a microservice A needs to make a request to another microservice B, microservice A needs to know the IP address and port of a specific instance where microservice B is hosted. More specifically, the communication layer between the microservices needs to know the IP addresses and ports of these microservices so that the requests between them can be routed appropriately. This is accomplished through *service discovery* (such as etcd, Consul, Hyperbahn, or ZooKeeper), which ensures that requests are routed to exactly where they are supposed to be sent and that (very importantly) they are only routed to healthy instances. Service discovery requires a *service registry*, which is a database that tracks all ports and IPs of all microservices across the ecosystem.

Dynamic Scaling and Assigned Ports

In microservice architecture, ports and IPs can (and do) change all of the time, especially as microservices are scaled and re-deployed (especially with a hardware abstraction layer like Apache Mesos). One way to approach the discovery and routing is to assign static ports (both frontend and backend) to each microservice.

Unless you have each microservice hosted on only one instance (which is highly unlikely), you'll need *load balancing* in place in various parts of the communication layer across the microservice ecosystem. Load balancing works, at a very high level, like this: if you have 10 different instances hosting a microservice, load-balancing

software (and/or hardware) will ensure that the traffic is distributed (balanced) across all of the instances. Load balancing will be needed at every location in the ecosystem in which a request is being sent to an application, which means that any large microservice ecosystem will contain many, many layers of load balancing. Commonly used load balancers for this purpose are Amazon Web Services Elastic Load Balancer, Netflix Eureka, HAProxy, and Nginx.

Summary of Layer 2: The Communication Layer

The communication layer (layer 2) of the microservice ecosystem contains:

- Network
- DNS
- Remote procedure calls (RPCs)
- Endpoints
- Messaging
- Service discovery
- Service registry
- Load balancing

Layer 3: The Application Platform

The *application platform* is the third layer of the microservice ecosystem and contains all of the internal tooling and services that are independent of the microservices. This layer is filled with centralized, ecosystem-wide tools and services that must be built in such a way that microservice development teams do not have to design, build, or maintain anything except their own microservices.

A good application platform is one with *self-service internal tools* for developers, a standardized *development process*, a centralized and automated *build and release system*, *automated testing*, a standardized and centralized *deployment solution*, and centralized *logging and microservice-level monitoring*. Many of the details of these elements are covered in later chapters, but we'll cover several of them briefly here to provide some introduction to the basic concepts.

Self-service internal development tools

Quite a few things can be categorized as *self-service internal development tools*, and which particular things fall into this category depends not only on the needs of the developers, but the level of abstraction and sophistication of both the infrastructure and the ecosystem as a whole. The key to determining which tools need to be built is

to first divide the realms of responsibility and then determine which tasks developers need to be able to accomplish in order to design, build, and maintain their services.

Within a company that has adopted microservice architecture, responsibilities need to be carefully delegated to different engineering teams. An easy way to do this is to create an engineering suborganization for each layer of the microservice ecosystem, along with other teams that bridge each layer. Each of these engineering organizations, functioning semi-independently, will be responsible for everything within their layer: TechOps teams will be responsible for layer 1, infrastructure teams will be responsible for layer 2, application platform teams will be responsible for layer 3, and microservice teams will be responsible for layer 4 (this is, of course, a very simplified view, but you get the general idea).

Within this organizational scheme, any time that an engineer working on one of the higher layers needs to set up, configure, or utilize something on one of the lower layers, there should be a self-service tool in place that the engineer can use. For example, the team working on messaging for the ecosystem should build a self-service tool so that if a developer on a microservice team needs to configure messaging for her service, she can easily configure the messaging without having to understand all of the intricacies of the messaging system.

There are many reasons to have these centralized, self-service tools in place for each layer. In a diverse microservice ecosystem, the average engineer on any given team will have no (or very little) knowledge of how the services and systems in other teams work, and there is simply no way they could become experts in each service and system while working on their own—it simply can't be done. Each individual developer will know almost nothing except her own service, but together, all of the developers working within the ecosystem will collectively know everything. Rather than trying to educate each developer about the intricacies of each tool and service within the ecosystem, build sustainable, easy-to-use user interfaces for every part of the ecosystem, and then educate and train them on how to use those. Turn everything into a black box, and document exactly how it works and how to use it.

The second reason to build these tools and build them well is that, in all honesty, you do not want a developer from another team to be able to make significant changes to your service or system, especially not one that could cause an outage. This is especially true and compelling for services and systems belonging to the lower layers (layer 1, layer 2, and layer 3). Allowing nonexperts to make changes to things within these layers, or requiring (or worse, expecting) them to become experts in these areas is a recipe for disaster. An example of where this can go terribly wrong is in configuration management: allowing developers on microservice teams to make changes to system configurations without having the expertise to do so can and will lead to large-scale production outages if a change is made that affects something other than their service alone.

The development cycle

When developers are making changes to existing microservices, or creating new ones, development can be made more effective by streamlining and standardizing the development process and automating away as much as possible. The details of standardizing the process of stable and reliable development itself are covered in Chapter 4, *Scalability and Performance*, but there are several things that need to be in place within the third layer of a microservice ecosystem in order for stable and reliable development to be possible.

The first requirement is a centralized *version control system* where all code can be stored, tracked, versioned, and searched. This is usually accomplished through something like GitHub, or a self-hosted git or svn repository linked to some kind of collaboration tool like Phabrictor, and these tools make it easy to maintain and review code.

The second requirement is a stable, efficient *development environment*. Development environments are notoriously difficult to implement in microservice ecosystems, due to the complicated dependencies each microservice will have on other services, but they are absolutely essential. Some engineering organizations prefer when all development is done locally (on a developer's laptop), but this can lead to bad deploys because it doesn't give the developer an accurate picture of how her code changes will perform in the production world. The most stable and reliable way to design a development environment is to create a mirror of the production environment (one that is not staging, nor canary, nor production) containing all of the intricate dependency chains.

Test, build, package, and release

The *test, build, package, and release steps* in between development and deployment should be standardized and centralized as much as possible. After the development cycle, when any code change has been committed, all the necessary tests should be run, and new releases should be automatically built and packaged. *Continuous integration* tooling exists for precisely this purpose, and existing solutions (like Jenkins) are very advanced and easy to configure. These tools make it easy to automate the entire process, leaving very little room for human error.

Deployment pipeline

The *deployment pipeline* is the process by which new code makes its way to production servers after the development cycle and following the test, build, package, and release steps. Deployment can quickly become very complicated in a microservice ecosystem, where hundreds of deployments per day are not out of the ordinary. Building tooling around deployment, and standardizing deployment practices for all development teams is often necessary. The principles of building stable and reliable

(production-ready) deployment pipelines are covered in detail in Chapter 3, *Stability and Reliability*.

Logging and monitoring

All microservices should have *microservice-level logging* of all requests made to the microservice (including all relevant and important information) and its responses. Due to the fast-paced nature of microservice development, it's often impossible to reproduce bugs in the code because it's impossible to reconstruct the state of the system at the time of failure. Good microservice-level logging gives developers the information they need to fully understand the state of their service at a certain time in the past or present. *Microservice-level monitoring* of all *key metrics* of the microservices is essential for similar reasons: accurate, real-time monitoring allows developers to always know the health and status of their service. Microservice-level logging and monitoring are covered in greater detail in ???.

Summary of Layer 3: The Application Platform Layer

The application platform layer (layer 3) of the microservice ecosystem contains:

- Self-service internal development tools
- Development environment
- Test, package, build, and release tools
- Deployment pipeline
- Microservice-level logging
- Microservice-level monitoring

Layer 4: Microservices

At the very top of the microservice ecosystem lies the *microservice layer* (*layer 4*). This layer is where the microservices—and anything specific to them—live, completely abstracted away from the lower infrastructure layers. Here they are abstracted from the hardware, from deployment, from service discovery, from load balancing, and from communication. The only things that are not abstracted away from the microservice layer are the configurations specific to each service for using the tools.

It is common practice in software engineering to centralize all application configurations so that the configurations for a specific tool or set of tools (like configuration management, resource isolation, or deployment tools) are all stored with the tool itself. For example, custom deployment configurations for applications are often stored not with the application code but with the code for the deployment tool. This

practice works well for monolithic architecture, and even for small microservice ecosystems, but in very large microservice ecosystems containing hundreds of microservices and dozens of internal tools (each with their own custom configurations), this practice becomes rather messy: developers on microservice teams are required to make changes to codebases of tools in the layers below, and oftentimes will forget where certain configurations live (or that they exist at all). To mitigate this problem, all microservice-specific configurations can live in the repository of the microservice and should be accessed there by the tools and systems of the layers below.

Summary of Layer 4: The Microservice Layer

The microservice layer (layer 4) of the microservice ecosystem contains:

- The microservices
- All microservice-specific configurations

Organizational Challenges

The adoption of microservice architecture resolves the most pressing challenges presented by monolithic application architecture. Microservices aren't plagued by the same scalability challenges, the lack of efficiency, or the difficulties in adopting new technologies: they are optimized for scalability, optimized for efficiency, optimized for developer velocity. In an industry where new technologies rapidly gain market traction, the pure organizational cost of maintaining and attempting to improve a cumbersome monolithic application is simply not practical. With these things in mind, it's hard to imagine why anyone would be reluctant to split a monolith into microservices, why anyone would be wary about building a microservice ecosystem from the ground up.

Microservices seem like a magical (and somewhat obvious) solution, but we know better than that. In *The Mythical Man-Month*, Frederick Brooks explained why there are no silver bullets in software engineering, an idea he summarized as follows: "There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity."

When we find ourselves presented with technology that promises to offer us drastic improvements, we need to look for the trade-offs. Microservices promise greater scalability and greater efficiency, but we know that those will come at a cost to some part of the overall system.

There are four especially significant trade-offs that come with microservice architecture. The first is the change in organizational structure that tends toward isolation

and poor cross-team communication—a consequence of the inverse of *Conway's Law*. The second is the dramatic increase in *technical sprawl*, sprawl that is extraordinarily costly not only to the entire organization but which also presents significant costs to each engineer. The third trade-off is the increased *ability of the system to fail*. The fourth is the *competition for engineering and infrastructure resources*.

The Inverse Conway's Law

The idea behind *Conway's Law* (named after programmer Melvin Conway in 1968) is this: that the architecture of a system will be determined by the communication and organizational structures of the company. The inverse of Conway's Law (which we'll call the *Inverse Conway's Law*) is also valid and is especially relevant to the microservice ecosystem: the organizational structure of a company is determined by the architecture of its product. Over 40 years after Conway's Law was first introduced, both it and its inverse still appear to hold true. Microsoft's organizational structure, if sketched out as if it were the architecture of a system, looks remarkably like the architecture of its products—the same goes for Google, for Amazon, and for every other large technology company. Companies that adopt microservice architecture will never be an exception to this rule.

Microservice architecture is comprised of a large number of small, isolated, independent microservices. The Inverse Conway's Law demands that the organizational structure of any company using microservice architecture will be made up of a large number of very small, isolated, and independent teams. The team structures that spring from this inevitably lead to siloing and sprawl, problems that are made worse every time the microservice ecosystem becomes more sophisticated, more complex, more concurrent, and more efficient.

Inverse Conway's Law also means that developers will be, in some ways, just like microservices: they will be able to do one thing, and (hopefully) do that one thing very well, but they will be isolated (in responsibility, in domain knowledge, and experience) from the rest of the ecosystem. When considered together, all of the developers *collectively* working within a microservice ecosystem will know everything there is to know about it, but individually they will be extremely specialized, knowing only the pieces of the ecosystem they are responsible for.

This poses an unavoidable organizational problem: even though microservices must be developed in isolation (leading to isolated, siloed teams), they don't live in isolation and must interact with one another seamlessly if the overall product is to function at all. This requires that these isolated, independently functioning teams work together closely and often—something that is difficult to accomplish, given that most team's goals and projects (codified in their team's objectives and key results, or OKRs) are specific to a particular microservice they are working on.

There is also a large communication gap between microservice teams and infrastructure teams that needs to be closed. Application platform teams, for example, need to build platform services and tools that all of the microservice teams will use, but gaining the requirements and needs from hundreds of microservice teams before building one small project can take months (even years). Getting developers and infrastructure teams to work together is not an easy task.

There's a related problem that arises thanks to Inverse Conway's Law, one that is only rarely found in companies with monolithic architecture: the difficulty of running an operations organization. With a monolith, an operations organization can easily be staffed and on call for the application, but this is very difficult to achieve with microservice architecture because it would require every single microservice to be staffed by both a development team *and* an operational team. Consequently, microservice development teams need to be responsible for the operational duties and tasks associated with their microservice. There is no separate ops org to take over the on call, no separate ops org responsible for monitoring: developers will need to be on call for their services.

Technical Sprawl

The second trade-off, *technical sprawl*, is related to the first. While Conway's Law and its inverse predict organizational sprawl and siloing for microservices, a second type of sprawl (related to technologies, tools, and the like) is also unavoidable in microservice architecture. There are many different ways in which technical sprawl can manifest. We'll cover a few of the most common ways here.

It's easy to see why microservice architecture leads to technical sprawl if we consider a large microservice ecosystem, one containing 1,000 microservices. Suppose each of these microservices is staffed by a development team of six developers, and each developer uses their own set of favorite tools, favorite libraries, and works in their own favorite languages. Each of these development teams has their own way of deploying, their own specified metrics to monitor and alert on, their own external libraries and internal dependencies they use, custom scripts to run on production machines, and so on.

If you have a thousand of these teams, this means that within one system there are a thousand ways to do one thing. There will be a thousand ways to deploy, a thousand libraries to maintain, a thousand different ways of alerting and monitoring and testing and handling outages. The only way to cut down on technical sprawl is through standardization at every level of the microservice ecosystem.

There's another kind of technical sprawl associated with language choice. Microservices infamously come with the promise of greater developer freedom, freedom to choose whichever languages and libraries one wants. This is possible in principle, and can be true in practice, but as a microservice ecosystem grows it often becomes

impractical, costly, and dangerous. To see why this can become a problem, consider the following scenario. Suppose we have a microservice ecosystem containing 200 services, and imagine that some of these microservices are written in Python, others in JavaScript, some in Haskell, a few in Go, and a couple more in Ruby, Java, and C++. For each internal tool, for each system and service within every layer of the ecosystem, libraries will have to be written for each one of these languages.

Take a moment to contemplate the sheer amount of maintenance and development that will have to be done in order for each language to receive the support it requires: it's extraordinary, and very few engineering organizations could afford to dedicate the engineering resources necessary to make it happen. It's more realistic to choose a small number of supported languages and ensure that all libraries and tools are compatible with and exist for these languages than to attempt to support a large number of languages.

The last type of technical sprawl we will cover here is technical debt, which usually refers to work that needs to be done because something was implemented in a way that got the job done quickly, but not in the best or most optimal way. Given that microservice development teams can churn out new features at a fast pace, technical debt often builds up quietly in the background. When outages happen, when things break, any work that comes out of an incident review will only rarely be the best overall solution: as far as microservice development teams are concerned, whatever fixes (or fixed) the problem quickly and in the moment was good enough, and any better solutions are pawned off to the future.

More Ways to Fail

Microservices are large, complex, distributed systems with many small, independent pieces that are constantly changing. The reality of working with complex systems of this sort is that individual components will fail, they will fail often, and they will fail in ways that nobody could have predicted. This is where the third trade-off comes into play: microservice architecture introduces more ways your system can fail.

There are ways to prepare for failure, to mitigate failures when they occur, and to test the limits and boundaries of both the individual components and the overall ecosystem, which I cover in ???. However, it is important to understand that no matter how many resiliency tests you run, no matter how many failures and catastrophe scenarios you've scoped out, you cannot escape the fact that the system *will* fail. You can only do your best to prepare for when it does.

Competition for Resources

Just like any other ecosystem in the natural world, competition for resources in the microservice ecosystem is fierce. Each engineering organization has finite resources: it has finite engineering resources (teams, developers) and finite hardware and infra-

structure resources (physical machines, cloud hardware, database storage, etc.), and each resource costs the company a great deal of money.

When your microservice ecosystem has a large number of microservices and a large and sophisticated application platform, competition between teams for hardware and infrastructure resources is inevitable: every service, every tool will be presented as equally important, its scaling needs presented as being of the highest priority.

Likewise, when application platform teams are asking for specifications and needs from microservice teams so that they can design their systems and tools appropriately, every microservice development team will argue that their needs are the most important and will be disappointed (and potentially very frustrated) if they are not included. This kind of competition for engineering resources can lead to resentment between teams.

The last kind of competition for resources is perhaps the most obvious one: the competition between managers, between teams, and between different engineering departments/organization for engineering headcount. Even with the increase in computer science graduates and the rise of developer bootcamps, truly great developers are difficult to find, and represent one of the most irreplaceable and scarce resources. When there are hundreds or thousands of teams that could use an extra engineer or two, every single team will insist that their team needs an extra engineer more than any of the other teams.

There is no way to avoid competition for resources, though there are ways to mitigate competition somewhat. The most effective seems to be organizing or categorizing teams in terms of their importance and criticality to the overall business, and then giving teams access to resources based on their priority or importance. There are downsides to this, because it tends to result in poorly staffed development tools teams, and in projects whose importance lies in shaping the future (such as adopting new infrastructure technologies) being abandoned.

CHAPTER 3

Stability and Reliability

A production-ready microservice is stable and reliable. Both individual microservices and the overall microservice ecosystem are constantly changing and evolving, and any efforts made to increase the stability and reliability of a microservice go a long way toward ensuring the health and availability of the overall ecosystem. In this chapter, different ways to build and run a stable and reliable microservice are explored, including standardizing the development process, building comprehensive deployment pipelines, understanding dependencies and protecting against their failures, building stable and reliable routing and discovery, and establishing appropriate deprecation and decommissioning procedures for old or outdated microservices and/or their endpoints.

Principles of Building Stable and Reliable Microservices

Microservice architecture lends itself to fast-paced development. The freedom offered by microservices means that the ecosystem will be in a state of continuous change, never static, always evolving. Features will be added every day, new builds will be deployed multiple times per day, and old technologies will be swapped for newer and better ones at an astounding pace. This freedom and flexibility gives rise to real, tangible innovation, but comes at a great cost.

Innovation, increased developer velocity and productivity, rapid technological advancement, and the ever-changing microservice ecosystem can all very quickly be brought to a screeching halt if any piece of the microservice ecosystem becomes unstable or unreliable. In some cases, all it takes to bring the entire business down is deploying a broken build or a build containing a bug to one business-critical microservice.

A *stable* microservice is one for which development, deployment, the adoption of new technologies, and the decommissioning or deprecation of other services do not give rise to instability across the larger microservice ecosystem. This requires putting measures into place to protect against the negative consequences that may be introduced by these types of changes. A *reliable* microservice is one that can be trusted by other microservices and by the overall ecosystem. Stability goes hand in hand with reliability because each stability requirement carries with it a reliability requirement (and vice versa): for example, stable deployment processes are accompanied by a requirement that each new deployment does not compromise the reliability of the microservice from the point of view of one of their clients or dependencies.

There are several things that can be done to ensure that a microservice is stable and reliable. A standardized *development cycle* can be implemented to protect against poor development practices. The *deployment* process can be designed so that changes to the code are forced to pass through multiple stages before being rolled out to all production servers. *Dependency* failures can be protected against. Health checks, proper routing, and circuit breaking can be built into the *routing and discovery* channels to handle anomalous traffic patterns. Finally, microservices and their endpoints can be *deprecated* and/or *decommissioned* without causing any failures for other microservices.

A Production-Ready Service Is Stable and Reliable

- It has a standardized development cycle.
- Its code is thoroughly tested through lint, unit, integration, and end-to-end testing.
- Its test, packaging, build, and release process is completely automated.
- It has a standardized deployment pipeline, containing staging, canary, and production phases.
- Its clients are known.
- Its dependencies are known, and there are backups, alternatives, fallbacks, and caching in place in case of failures.
- It has stable and reliable routing and discovery in place.

The Development Cycle

The stability and reliability of a microservice begins with the individual developer who is contributing code to the service. The majority of outages and microservice failures are caused by bugs introduced into the code that were not caught in the development phase, in any of the tests, or at any step in the deployment process. Miti-

gating and resolving these outages and failures usually entails nothing more than rolling back to the latest stable build, reverting whatever commit contained the bug, and re-deploying a new (bug-less) version of the code.

The True Cost of Unstable and Unreliable Development

A microservice ecosystem is not the Wild West. Every outage, every incident, and every bug can and will cost the company thousands (if not millions) of dollars in engineering hours and lost revenue. Safeguards need to be in place during the development cycle (and, as we will see, in the deployment pipeline) to catch every bug before it hits production.

A stable and reliable development cycle has several steps (Figure 3-1).

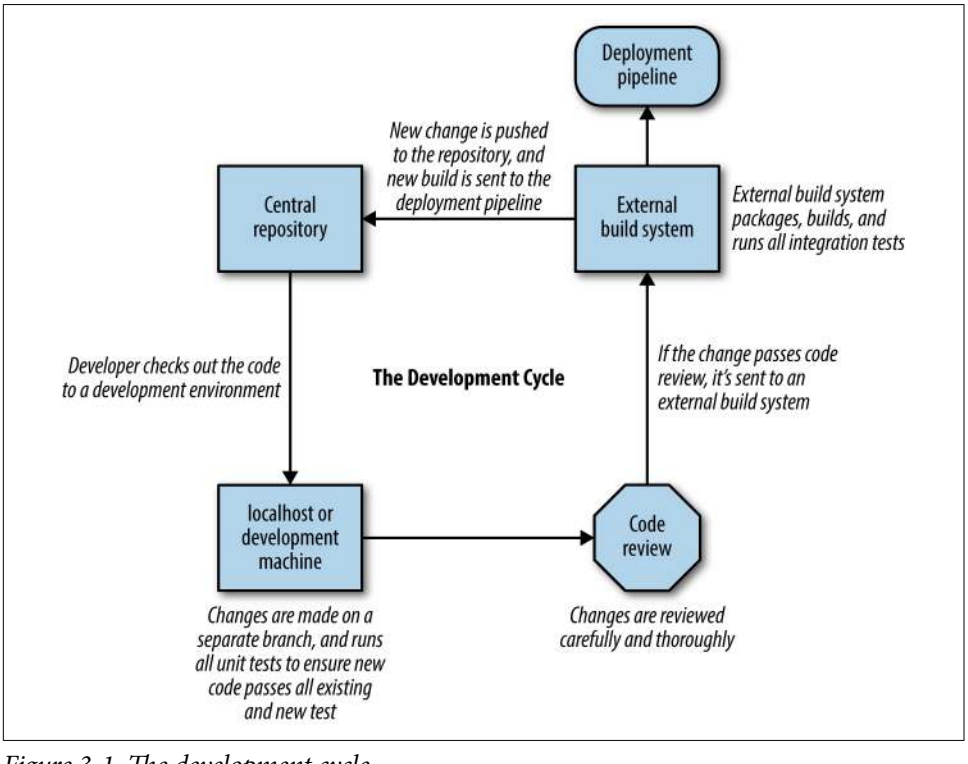

Figure 3-1. The development cycle

First, the developer makes a change to the code. This will usually begin with checking a copy of the code out from a central repository (usually using git or svn), creating an individual branch where they will make changes, adding their changes to their branch, and running any unit and integration tests. This stage of development can

happen anywhere: locally on a developer's laptop or on a server in a development environment. A reliable development environment—one that accurately mirrors the production world—is key, especially if testing the service in question requires making requests to other microservices or reading or writing data to a database.

Once the code has been committed to the central repository, the second step consists in having the change(s) reviewed carefully and thoroughly by other engineers on the team. If all reviewers have approved the change(s), and all lint, unit, and integration tests have passed on a new build, the change can be merged into the repository (see ???, for more on lint, unit, and integration tests). Then, and only then, can the new change be introduced into the deployment pipeline.

Test Before Code Review

One way to ensure that all bugs are caught before they hit production is to run all lint, unit, integration, and end-to-end tests *before* the code review phase. This can be accomplished by having developers work on a separate branch, kicking off all tests on that branch as soon as the developer submits it for code review, and then only allowing it to reach code review (or only allowing it to be built) *after* it successfully passes all tests.

As mentioned in the section on layer 4 of the microservice ecosystem in Chapter 1, *Microservices*, a lot happens in between the development cycle and the deployment pipeline. The new release needs to be packaged, built, and thoroughly tested before reaching the first stage of the deployment pipeline.

The Deployment Pipeline

There is a great deal of room for human error in microservice ecosystems, especially where deployment practices are concerned, and (as I mentioned earlier) the majority of outages in large-scale production systems are caused by bad deployments. Consider the organizational sprawl that accompanies the adoption of microservice architecture and what it entails for the deployment process: you have, at the very least, dozens (if not hundreds or thousands) of independent, isolated teams who are deploying changes to their microservices on their own schedules, and often without cross-team coordination between clients and dependencies. If something goes wrong, if a bug is introduced into production, or if a service is temporarily unavailable during deployment, then the entire ecosystem can be negatively affected. To ensure that things go wrong with less frequency, and that any failures can be caught before being rolled out to all production servers, introducing a standardized *deployment pipeline* across the engineering organization can help ensure stability and reliability across the ecosystem.

I refer to the deployment process here as a "pipeline" because the most trustworthy deployments are those that have been required to pass a set of tests before reaching production servers. We can fit three separate stages or phases into this pipeline (Figure 3-2): first, we can test a new release in a *staging* environment; second, if it passes the staging phase, we can deploy it to a small *canary* environment, where it will serve 5%–10% of production traffic; and third, if it passes the canary phase, we can slowly roll it out to *production* servers until it has been deployed to every host.

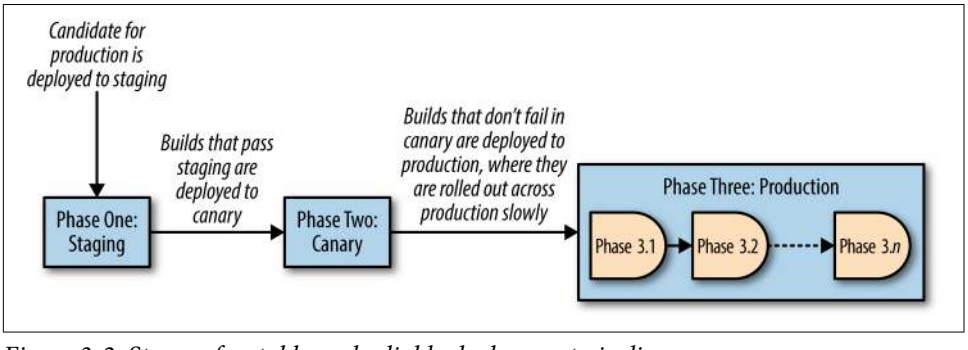

Figure 3-2. Stages of a stable and reliable deployment pipeline

Staging

Any new release can first be deployed to a *staging* environment. A staging environment should be an exact copy of the production environment: it is a reflection of the state of the real world, but without real traffic. Staging environments usually aren't running at the same scale as production (i.e., they typically aren't run with the same number of hosts as production, a phenomenon also known as *host parity*), because running what would amount to two separate ecosystems can present a large hardware cost to the company. However, some engineering organizations may determine that the only way to accurately copy the production environment in a stable and reliable way is to build an identical staging environment with host parity.

For most engineering organizations, determining the hardware capacity and scale of the staging environment as a percentage of production is usually accurate enough. The necessary staging capacity can be determined by the method we will use to test the microservice within the staging phase. To test in the staging environment, we have several options: we can run mock (or recorded) traffic through the microservice; we can test it manually by hitting its endpoints and evaluating its responses; we can run automated unit, integration, and other specialized tests; or we can test each new release with any combination of these methods.

Treat Staging and Production as Separate Deployments of the Same Service

You may be tempted to run staging and production as separate services and store them in separate repositories. This *can* be done successfully, but it requires that changes be synchronized across both services and repositories, including configuration changes (which are often forgotten about). It's much easier to treat staging and production as separate "deployments" or "phases" of the same microservice.

Even though staging environments *are* testing environments, they differ from both the development phase and the development environment in that a release that has been deployed to staging is a release that is a *candidate for production*. A candidate for production must have already successfully passed lint tests, unit tests, integration tests, and code review before being deployed to a staging environment.

Deploying to a staging environment should be treated by developers with the same seriousness and caution as deploying to production. If a release is successfully deployed to staging, it can be automatically deployed to canaries, which *will* be running production traffic.

Setting up staging environments in a microservice ecosystem can be difficult, due to the complexities introduced by dependencies. If your microservice depends on nine other microservices, then it relies on those dependencies to give accurate responses when requests are sent and reads or writes to the relevant database(s) are made. As a consequence of these complexities, the success of a staging environment hinges on the way staging is standardized across the company.

Full staging

There are several ways that the staging phase of the deployment pipeline can be configured. The first is *full staging* (Figure 3-3), where a separate staging ecosystem is running as a complete mirror copy of the entire production ecosystem (though not necessarily with host parity). Full staging still runs on the same core infrastructure as production, but there are several key differences. Staging environments of the services are, at the very least, made accessible to other services by staging-specific frontend and backend ports. Importantly, staging environments in a full-staging ecosystem communicate *only with the staging environments of other services*, and never send any requests or receive any responses from any services running in production (which means sending traffic to production ports from staging is off limits).

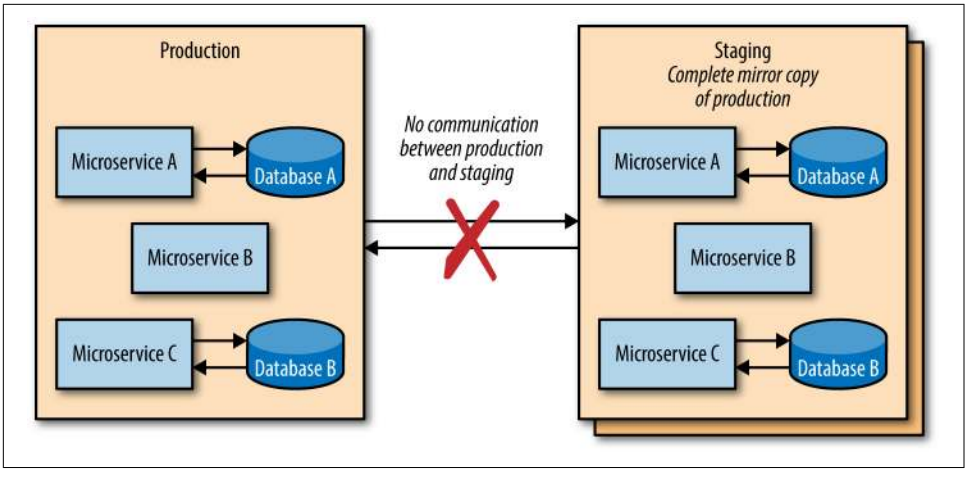

Figure 3-3. Full staging

Full staging requires every microservice to have a fully functional staging environment that other microservices can communicate with when new releases are deployed. Communicating with other microservices within the staging ecosystem can be accomplished either by writing specific tests that are kicked off when a new build is deployed to the staging environment, or as mentioned, by running old recorded production traffic or mock traffic through the service being deployed along with all upstream and downstream dependencies.

Full staging also requires careful handling of test data: staging environments should *never* have write access to any production databases, and granting read access to production databases is discouraged as well. Because full staging is designed to be a complete mirror copy of production, every microservice staging environment should contain a separate test database that it can read from and write to.

Risks of Full Staging

Caution needs to be taken when implementing and deploying full staging environments, because new releases of services will almost always be communicating with other new releases of any upstream and downstream dependencies—this may not be an accurate reflection of the real world. Engineering organizations may need to require teams to coordinate and/or schedule deployments to staging to avoid the deployment of one service breaking the staging environment for all other related services.

Partial staging

The second type of staging environment is known as *partial staging*. As the name suggests, it is not a complete mirror copy of the production environment. Rather, each microservice has its own staging environment, which is a pool of servers with (at the very least) staging-specific frontend and backend ports, and when new builds are introduced into the staging phase, they communicate with the upstream clients and downstream dependencies that are running in production (Figure 3-4).

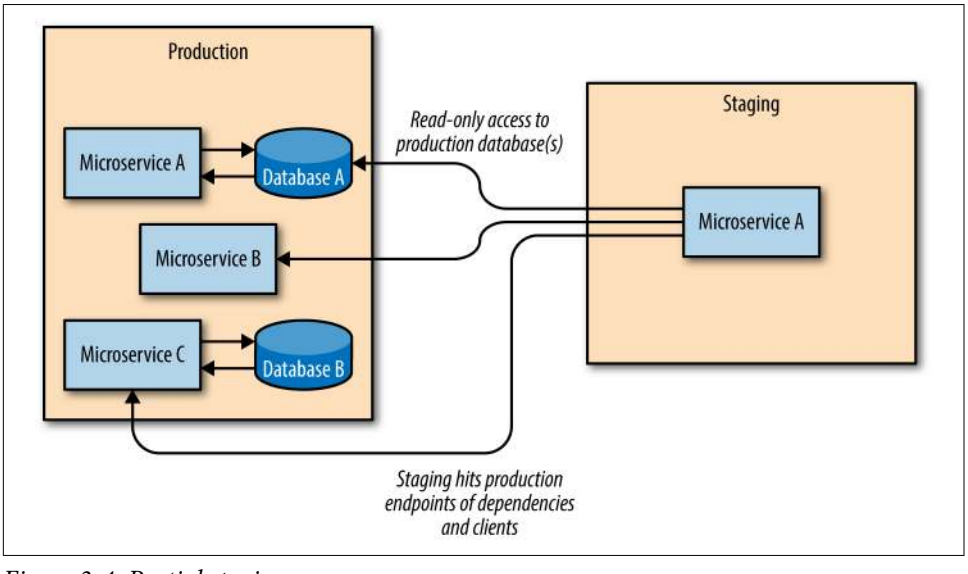

Figure 3-4. Partial staging

Partial staging deployments should hit all production endpoints of a microservice's clients and dependencies to mimic the state of the actual world as accurately as possible. Specific staging tests will need to be written and run to accomplish this, and every new feature added should probably be accompanied by at least one additional staging test to ensure that it is tested thoroughly.

Risks of Partial Staging

Because microservices with partial staging environments communicate with production microservices, extreme care must be taken. Even though partial staging is restricted to read-only requests, production services can easily be taken down by bad staging deploys that send bad requests and/or overload production services with too many requests.

These types of staging environments should also be restricted to read-only database access: a staging environment should never write to a production database. However,

some microservices may be very write-heavy, and testing the write functionality of a new build will be essential. The most common way of doing this is to mark any data written by a staging environment as *test data* (this is known as *test tenancy*), but the safest way to do this is to write to a separate test database, since giving write access to a staging environment still runs the risk of altering real-world data. See Table 3-1 for a comparison of full and partial staging environments.

Table 3-1. Full versus partial staging environments

	Full staging	Partial staging
Complete copy of production environment	Yes	No
Separate staging frontend and backend ports	Yes	Yes
Access to production services	No	Yes
Read access to production databases	No	Yes
Write access to production databases	No	Yes
Requires automated rollbacks	No	Yes

Staging environments (full or partial) should have dashboards, monitoring, and logging just like production environments—all of which should be set up identically to the dashboards, monitoring, and logging of the production environment of the microservice (see ???). The graphs for all key metrics can be kept on the same dashboard as all production metrics, though teams may opt to have separate dashboards for each part of the deployment process: a staging dashboard, a canary dashboard, and a production dashboard. Depending on how dashboards are configured, it may be best to keep all graphs for all deployments on one dashboard and to organize them by deployment (or by metric). Regardless of how a team decides to set up their dashboards, the goal of building good and useful production-ready dashboards should not be forgotten: the dashboard(s) of a production-ready microservice should make it easy for an outsider to quickly determine the health and status of the service.

Monitoring and logging for the staging environment should be identical to the monitoring and logging of the staging and production deployments so that any failures of tests and errors in new releases that are deployed to staging will be caught before they move to the next phase of the deployment pipeline. It's extremely helpful to set up alerts and logs so that they are differentiated and separated by deployment type, ensuring that any alerts triggered by failures or errors will specify which environment is experiencing the problem, making debugging, mitigation, and resolution of any bugs or failures rather easy and straightforward.

The purpose of a staging environment is to catch any bugs introduced by code changes before they affect production traffic. When a bug is introduced by the code, it will usually be caught in the staging environment (if it is set up correctly). Automated rollbacks of bad deploys are a necessity for partial staging environments (though

are not required for full staging environments). Establishing when to revert to a previous build should be determined by various thresholds on the microservice's key metrics.

Since partial staging requires interacting with microservices running in production, bugs introduced by new releases deployed to a partial staging environment can bring down other microservices that are running in production. If there aren't any automated rollbacks in place, mitigating and resolving these problems needs to be done manually. Any steps of the deployment process that need manual intervention are points of failure not only for the microservice itself, but for the entire microservice ecosystem.

The last question a microservice team needs to answer when setting up a staging environment is how long a new release should run on staging before it can be deployed to canary (and, after that, to production). The answer to this question is determined by the staging-specific tests that are run on staging: a new build is ready to move to the next step of the deployment process as soon as all tests have passed without failing.

Canary

Once a new release has successfully been deployed to staging and passed all required tests, the build can be deployed to the next stage in the deployment pipeline: the *canary* environment. The unique name for this environment comes from a tactic used by coal miners: they'd bring canaries with them into the coal mines to monitor the levels of carbon monoxide in the air; if the canary died, they knew that the level of toxic gas in the air was high, and they'd leave the mines. Sending a new build into a canary environment serves the same purpose: deploy it to a small pool of servers running production traffic (around 5%–10% of production capacity), and if it survives, deploy to the rest of the production servers.

Canary Traffic Distribution

If the production service is deployed in multiple different datacenters, regions, or cloud providers, then the canary pool should contain servers in each of these in order to accurately sample production.

Since a canary environment serves production traffic, it should be considered part of production. It should have the same backend and frontend ports, and canary hosts should be chosen at random from the pool of production servers to ensure accurate sampling of production traffic. Canaries can (and should) have full access to production services: they should hit all production endpoints of upstream and downstream

dependencies, and they should have both read and write access to any databases (if applicable).

As with staging, the dashboards, monitoring, and logging should be the same for canaries as for production. Alerts and logs should be differentiated and labeled as coming from the canary deployment so that developers can easily mitigate, debug, and resolve any problems.

Separate Ports for Canaries and Production

Allocating separate frontend and backend ports for canaries and production so that traffic can be directed deliberately may seem like a good idea, but unfortunately separating out the traffic in this fashion defeats the purpose of canaries: to randomly sample production traffic on a small pool of servers to test a new release.

Automated rollbacks absolutely need to be in place for canaries: if any known errors occur, the deployment system needs to automatically revert to the last known stable version. Remember, canaries are serving production traffic, and any problems that happen are affecting the real world.

How long should a new release sit in the canary pool until developers can be satisfied that it is ready for production? This can be minutes, hours, or even days, and the answer is determined by the microservice's traffic patterns. The traffic of every microservice is going to have some sort of pattern, no matter how strange your microservice or business may be. A new release should not leave the canary stage of deployment until a full traffic cycle has been completed. How a "traffic cycle" is defined needs to be standardized across the entire engineering organization, but the duration and requirements of the traffic cycle may need to be created on a service-by-service basis.

Production

Production is the real world. When a build has successfully made it through the development cycle, survived staging, and lived through the coal mines of the canary phase, it is ready to be rolled out to the production deployment. At this point in the deployment pipeline—the very last step—the development team should be completely confident in the new build. Any errors in the code should have been discovered, mitigated, and resolved before making it this far.

Every build that makes it to production should be completely stable and reliable. A build being deployed to production should have already been thoroughly tested, and a build should *never* be deployed to production until it has made it through the staging and canary phases without any issues. Deploying to production can be done in one fell swoop after the build has lived through the canaries, or it can be gradually

rolled out in stages: developers can choose to roll out to production by percentage of hardware (e.g., first to 25% of all servers, then to 50%, then 75%, and finally 100%), or by datacenter, or by region, or by country, or any mixture of these.

Enforcing Stable and Reliable Deployment

By the time a new candidate for production has made it through the development process, has survived the staging environment, and has been deployed to the canary phase successfully, the chances of it causing a major outage are very slim, because most bugs in the code will have been caught before the candidate for production is rolled out to production. This is precisely why having a comprehensive deployment pipeline is essential for building a stable and reliable microservice.

For some developers, the delay introduced by the deployment pipeline might seem like an unnecessary burden because it delays their code changes and/or new features from being deployed straight to production minutes after they have been written. In reality, the delay introduced by the phases of the deployment pipeline is very short and easily customizable, but sticking to the standardized deployment process needs to be enforced to ensure reliability. Deploying to a microservice multiple times per day can (and does) compromise the stability and reliability of the microservice and any other services within its complex dependency chain: a microservice that is changing every few hours is rarely a stable or reliable microservice.

Developers may be tempted to skip the staging and canary phases of the deployment process and deploy a fix straight to production if, for example, a serious bug is discovered in production. While this solves the problem quickly, can *possibly* save the company from losing revenue, and can prevent dependencies from experiencing outages, allowing developers to deploy straight to production should be reserved only for the most severe outages. Without these restrictions in place, there is always the unfortunate possibility of abusing the process and deploying straight to production: for most developers, every code change, every deploy is important and may seem important enough to bypass staging and canary, compromising the stability and reliability of the entire microservice ecosystem. When failures occur, development teams should instead be encouraged to always roll back to the latest stable build of the microservice, which will bring the microservice back to a known (and reliable) state, which can run in production without any issues while the team works to discover the root cause of the failure that occurred.

Hotfixes Are an Anti-Pattern

When a deployment pipeline is in place, there should never be any direct deployment to production unless there is an emergency, but even this should be discouraged. Bypassing the initial phases of the deployment pipeline often introduces new bugs into production, as emergency code fixes run the risk of not being properly tested. Rather than deploying a hotfix straight to production, developers should roll back to the latest stable build if possible.

Stable and reliable deployment isn't limited only to following the deployment pipeline, and there are several cases in which blocking a particular microservice from deploying can increase availability across the ecosystem.

If a service isn't meeting their SLAs (see ???), all deployment can be postponed if the downtime quota of the service has been used up. For example, if a service has an SLA promising 99.99% availability (allowing 4.38 minutes of downtime each month), but has been unavailable for 12 minutes in one month, then new deployments of that microservice can be blocked for the next three months, ensuring that it meets its SLA. If a service fails load testing (see ???), then deployment to production can be locked until the service is able to appropriately pass any necessary load tests. For business-critical services, whose outages would stop the company from functioning properly, it can at times be necessary to block deployment if they do not meet the production-readiness criteria established by the engineering organization.

Dependencies

The adoption of microservice architecture is sometimes driven by the idea that microservices can be built and run in isolation, as fully independent and replaceable components of a larger system. This is true in principle, but in the real world, every microservice has *dependencies*, both upstream and downstream. Every microservice will receive requests from *clients* (other microservices) that are counting on the service to perform as expected and to live up to its SLAs, as well as downstream dependencies (other services) that it will depend on to get the job done.

Building and running production-ready microservices requires developers to plan for dependency failures, to mitigate them, and to protect against them. Understanding a service's dependencies and planning for their failures is one of the most important aspects of building a stable and reliable microservice.

To understand how important this is, let's consider an example microservice called *receipt-sender*, whose SLA is four-nines (promising 99.99% availability to upstream clients). Now, *receipt-sender* depends on several other microservices, including one called *customers* (a microservice that handles all customer information), and one called *orders* (a microservice that handles information about the orders each cus-

tomer places). Both *customers* and *orders* depend on other microservices: *customers* depends on yet another microservice we'll call *customers-dependency*, and *orders* on one we'll refer to as *orders-dependency*. The chances that *customers-dependency* and *orders-dependency* have dependencies of their own are very high, so the dependency graph for *receipt-sender* quickly becomes very, very complicated.

Since *receipt-sender* wants to protect its SLA and provide 99.99% uptime to all of its clients, its team needs to make sure that the SLAs of all downstream dependencies are strictly adhered to. If the SLA of *receipt-sender* depends on *customers* being available 99.99% of the time, but the actual uptime of *customers* is only 89.99% of the time, the availability of *receipt-sender* is compromised and is now only 89.98%. Each one of the dependencies of *receipt-sender* can suffer the same hit to their availability if any of the dependencies in the dependency chain do not meet their SLAs.

A stable and reliable microservice needs to mitigate dependency failures of this sort (and yes, not meeting an SLA is a failure!). This can be accomplished by having backups, fallbacks, caching, and/or alternatives for each dependency just in case they fail.

Before dependency failures can be planned for and mitigated, the dependencies of a microservice must be known, documented, and tracked. Any dependency that could harm a microservice's SLA needs to be included in the architecture diagram and documentation of the microservice (see Chapter 7, *Documentation and Understanding*) and should be included on the service's dashboard(s) (see ???). In addition, all dependencies should be tracked by automatically creating dependency graphs for each service, which can be accomplished by implementing a distributed tracking system across all microservices in the organization.

Once all of the dependencies are known and tracked, the next step is to set up backups, alternatives, fallbacks, or caching for each dependency. The right way to do this is completely dependent on the needs of the service. For example, if the functionality of a dependency can be filled by calling the endpoint of another service, then failure of the primary dependency should be handled by the microservice so that requests are sent to the alternative instead. If requests that need to be sent to the dependency can be held in a queue when the dependency is unavailable, then a queue should be implemented. Another way to handle dependency failures is to put caching for the dependency into place within the service: cache any relevant data so that any failures will be handled gracefully.

The type of cache most often used in these cases is a *Least Recently Used* (LRU) cache, in which relevant data is kept in a queue, and where any unused data is deleted when the cache's queue fills up. LRU caches are easy to implement (often a single line of code for each instantiation), efficient (no expensive network calls need to be made), performant (the data is immediately available), and do a decent job of mitigating any dependency failures. This is known as *defensive caching*, and it is useful for protecting a microservice against the failures of its dependencies: cache the information your

microservice gets from its dependencies, and if the dependencies go down, the availability of your microservice will be unaffected. Implementing defensive caching isn't necessary for every single dependency, but if a specific dependency or set of dependencies is or are unreliable, defensive caching will prevent your microservice from being harmed.

Routing and Discovery

Another aspect of building stable and reliable microservices is to ensure that communication and interaction between microservices is itself stable and reliable, which means that layer 2 (the communication layer) of the microservice ecosystem (see Chapter 1, *Microservices*) must be built to perform in a way that protects against harmful traffic patterns and maintains trust across the ecosystem. The relevant parts of the communication layer for stability and reliability (aside from the network itself) are service discovery, service registry, and load balancing.

The *health* of a microservice at both the host level and the service level as a whole should always be known. This means that *health checks* should be running constantly so that a request is never sent to an unhealthy host or service. Running health checks on a separate channel (not used for general microservice communication) is the easiest way to ensure that health checks aren't ever compromised by something like a clogged network. Hardcoding "200 OK" responses on a */health* endpoint for health checks isn't ideal for every microservice either, though it may be sufficient for most. Hardcoded responses don't tell you much except that the microservice was started on the host semi-successfully: any */health* endpoint of a microservice should give a useful, accurate response.

If an instance of a service on a host is unhealthy, the load balancers should no longer route traffic to it. If a microservice as a whole is unhealthy (with all health checks failing on either a certain percentage of hosts or all hosts in production), then traffic should no longer be routed to that particular microservice until the problems causing the health checks to fail are resolved.

However, health checks shouldn't be the only determining factor in whether or not a service is healthy. A large number of unhandled exceptions should also lead to a service being marked unhealthy, and *circuit breakers* should be put into place for these failures so that if a service experiences an abnormal amount of errors, no more requests will be sent to the service until the problem is resolved. The key in stable and reliable routing and discovery is this: preserve the microservice ecosystem by preventing bad actors from serving production traffic and accepting requests from other microservices.

Deprecation and Decommissioning

One often-forgotten, often-ignored cause of instability and unreliability in microservice ecosystems is the *deprecation or decommissioning* of a microservice or one of its API endpoints. When a microservice is no longer in use or is no longer supported by a development team, its decommissioning should be undertaken carefully to ensure that no clients will be compromised. The deprecation of one or more of a microservice's API endpoints is even more common: when new features are added or old ones removed, the endpoints often change, requiring that client teams are updated and any requests made to the old endpoints are switched to new endpoints (or removed entirely).

In most microservice ecosystems, deprecation and decommissioning is more of a sociological problem within the engineering organization than a technical one, making it all the more difficult to address. When a microservice is about to be decommissioned, its development team needs to take care to alert all client services and advise them on how to accommodate the loss of their dependency. If the microservice being decommissioned is being replaced by another new microservice, or if the functionality of the microservice is being built into another existing microservice, then the team should help all clients update their microservices to send requests to the new endpoints. Deprecation of an endpoint follows a similar process: the clients must be alerted, and either given the new endpoint or advised on how to account for the loss of the endpoint entirely. In both deprecation and decommissioning, monitoring plays a critical role: endpoints will need to be monitored closely *before* the service or endpoint is completely decommissioned and/or deprecated to check for any requests that might still be sent to the outdated service or endpoint.

Conversely, failing to properly deprecate an endpoint or decommission a microservice can also have disastrous effects on the microservice ecosystem. This happens more often than developers would care to admit. In an ecosystem containing hundreds or thousands of microservices, developers are often shifted between teams, priorities change, and both microservices and technologies are swapped out for newer, better ones all of the time. When these old microservices or technologies are left to run, without any (or much) involvement, oversight, or monitoring, any failures will go unnoticed, and any failure that is noticed may not be resolved for a long period of time. If a microservice is going to be left to fend for itself, it risks compromising its clients in case of an outage—such microservices should be decommissioned rather than abandoned.

Nothing is more disruptive to a microservice than the complete loss of one of its dependencies. Nothing causes more instability and unreliability than the sudden, unexpected failure of one of its dependencies, even if the failure was planned for by another team. The importance of stable and reliable decommissioning and deprecation can honestly not be emphasized enough.

Evaluate Your Microservice

Now that you have a better understanding of stability and reliability, use the following list of questions to assess the production-readiness of your microservice(s) and microservice ecosystem. The questions are organized by topic, and correspond to the sections within this chapter.

The Development Cycle

- Does the microservice have a central repository where all code is stored?
- Do developers work in a development environment that accurately reflects the state of production (e.g., that accurately reflects the real world)?
- Are there appropriate lint, unit, integration, and end-to-end tests in place for the microservice?
- Are there code review procedures and policies in place?
- Is the test, packaging, build, and release process automated?

The Deployment Pipeline

- Does the microservice ecosystem have a standardized deployment pipeline?
- Is there a staging phase in the deployment pipeline that is either full or partial staging?
- What access does the staging environment have to production services?
- Is there a canary phase in the deployment pipeline?
- Do deployments run in the canary phase for a period of time that is long enough to catch any failures?
- Does the canary phase accurately host a random sample of production traffic?
- Are the microservice's ports the same for canary and production?
- Are deployments to production done all at the same time, or incrementally rolled out?
- Is there a procedure in place for skipping the staging and canary phases in case of an emergency?

Dependencies

- What are this microservice's dependencies?
- What are its clients?

- How does this microservice mitigate dependency failures?
- Are there backups, alternatives, fallbacks, or defensive caching for each dependency?

Routing and Discovery

- Are health checks to the microservice reliable?
- Do health checks accurately reflect the health of the microservice?
- Are health checks run on a separate channel within the communication layer?
- Are there circuit breakers in place to prevent unhealthy microservices from making requests?
- Are there circuit breakers in place to prevent production traffic from being sent to unhealthy hosts and microservices?

Deprecation and Decommissioning

- Are there procedures in place for decommissioning a microservice?
- Are there procedures in place for deprecating a microservice's API endpoints?

CHAPTER 4

Scalability and Performance

A production-ready microservice is scalable and performant. A scalable, performant microservice is one that is driven by efficiency, one that can not only handle a large number of tasks or requests at the same time, but can handle them efficiently and is prepared for tasks or requests to increase in the future. In this chapter, the essential components of microservice scalability and performance are covered, including understanding the qualitative and quantitative growth scales, hardware efficiency, identification of resource requirements and bottlenecks, capacity awareness and planning, scalable handling of traffic, the scaling of dependencies, task handling and processing, and scalable data storage.

Principles of Microservice Scalability and Performance

Efficiency is of the utmost importance in real-world, large-scale distributed systems architecture, and microservice ecosystems are no exception to this rule. It's easy to quantify the efficiency of a single system (like a monolithic application), but evaluating the efficiency and achieving greater efficiency in a large ecosystem of microservices, where tasks are sharded out between hundreds (if not thousands) of small services, is incredibly difficult. It's also bounded by the laws of computer architecture and distributed systems, which place limits on the efficiency of large-scale, complex distributed systems: the more distributed your system, and the more microservices you have in place within that system, the less of a difference the efficiency of one microservice will have on the entire system. Standardization of principles that will increase overall efficiency becomes a necessity. Two of our production-readiness standards—*scalability* and *performance*—help to achieve this overall efficiency, and increase the availability of the microservice ecosystem.

Scalability and performance are uniquely intertwined because of the effects they have on the efficiency of each microservice and the ecosystem as a whole. As we saw in

Chapter 1, *Microservices*, in order to build a scalable application, we need to design for concurrency and partitioning: concurrency allows each task to be broken up into smaller pieces, while partitioning is essential for allowing these smaller pieces to be processed in parallel. So, while *scalability* is related to how we divide and conquer the processing of tasks, *performance* is the measure of how efficiently the application processes those tasks.

In a growing, thriving microservice ecosystem, where traffic is increasing steadily, each microservice needs to be able to scale with the entire system without suffering from performance problems. To ensure that our microservices are scalable and performant, we need to require several things of each microservice. We need to understand its *growth scale*, both quantitative and qualitative, so that we can prepare for expected growth. We need to use our *hardware resources efficiently*, be aware of *resource bottlenecks and requirements*, and do appropriate *capacity planning*. We need to ensure that a microservice's *dependencies will scale* with it. We need to *manage traffic* in a scalable and performant way. We need to *handle and process tasks* in a performant manner. Last but not least, we need to *store data in a scalable way*.

A Production-Ready Service Is Scalable and Performant

- Its qualitative and quantitative growth scales are known.
- It uses hardware resources efficiently.
- Its resource bottlenecks and requirements have been identified.
- Capacity planning is automated and performed on a scheduled basis.
- Its dependencies will scale with it.
- It will scale with its clients.
- Its traffic patterns are understood.
- Traffic can be re-routed in case of failures.
- It is written in a programming language that allows it to be scalable and performant.
- It handles and processes tasks in a performant manner.
- It handles and stores data in a scalable and performant way.

Knowing the Growth Scale

Determining *how* a microservice scales (at a very high level) is the first step toward understanding how to build and maintain a scalable microservice. There are two aspects to knowing the *growth scale* of a microservice, and they both play important roles in understanding and planning for the scalability of a service. The first is the

qualitative growth scale, which comes from understanding where the service fits into the overall microservice ecosystem and which key high-level business metrics it will be affected by. The second is the *quantitative growth scale*, which is, as its name suggests, a well-defined, measurable, and quantitative understanding of how much traffic a microservice can handle.

The Qualitative Growth Scale

The natural tendency when trying to determine the growth scale of a microservice is to phrase the growth scale in terms of *requests per second* (RPS) or *queries per second* (QPS) that the service can support, then predicting how many RPS/QPS will be made to the service in the future. The term "requests per second" is generally used when talking about microservices, and "queries per second" when talking about databases or microservices that return data to clients, though in many cases they are interchangeable. This is very important information, but it's useless without additional context—specifically, without the context of where the microservice fits into the overall picture.

In most cases, information about the RPS/QPS a microservice can support is determined by the state of the microservice at the time the growth scale is initially calculated: if the growth scale is calculated by only looking at the current levels of traffic and how the microservice handles the current traffic load, making any inferences about how much traffic the microservice can handle in the future runs the risk of being misguided. There are several approaches one could take to get around this problem, including load testing (testing the microservice with higher loads of traffic), which can present a more accurate picture of the scalability of the service, and analyzing historical traffic data to see how the traffic level grows over time. But there's something very key missing here, something that is an inherent property of microservice architecture—namely, that microservices do not live alone but as part of a larger ecosystem.

This is where the *qualitative growth scale* comes in. Qualitative growth scales allow the scalability of a service to tie in with higher-level business metrics: a microservice may, for example, scale with the number of users, with the number of people who open a phone application ("eyeballs"), or with the number of orders (for a food delivery service). These metrics, these qualitative growth scales, aren't tied to an individual microservice but to the overall system or product(s). At the business level, the organization will have, for the most part, some idea of how these metrics will change over time. When these higher-level business metrics are communicated to engineering teams, developers can interpret them as they relate to their respective microservices: if one of their microservices is part of the order flow for a food delivery service, they will know that any metrics related to the number of orders expected in the future will tell them what kind of traffic their service should expect.

When I ask microservice development teams if they know the growth scale of their service, the usual response is, "It can handle *x* requests per second." My follow-up questions are always geared toward discovering where the service in question fits into the overall product: When are requests made? Is it one request per trip? One request each time someone opens the app? One request every time a new user signs up for our product? When these context-filling questions are answered, the growth scale becomes clear—and useful. If the number of requests made to the service is directly linked to the number of people who open a phone application, then the service scales with eyeballs, and we can plan for scaling the service by predicting how many people will be opening the application. If the number of requests made to the service is determined by the number of people who order delivery food, then the service scales with deliveries, and we can plan and predict for scaling our service by using higher-level business metrics about how many future deliveries are predicted.

There are exceptions to the rules of qualitative growth scales, and determining an appropriate qualitative growth scale can become very complicated the further down the stack the service is found. Internal tools tend to suffer from these complications, and yet they tend to be so business-critical that if they aren't scalable, the rest of the organization quickly hits scalability challenges. It's not easy to put the growth scale of a service like a monitoring or alerting platform in terms of business metrics (users, eyeballs, etc.), so platform and/or infrastructure organizations need to determine accurate growth scales for their services in terms of their customers (developers, services, etc.) and their customers' specifications. Internal tools can scale with, for example, number of deployments, number of services, number of logs aggregated, or gigabytes of data. These are more complicated because of the inherent difficulty in predicting these numbers, but they must be just as straightforward and predictable as the growth scales of microservices higher in the stack.

The Quantitative Growth Scale

The second part of knowing the growth scale is determining its quantitative aspects, which is where RPS/QPS and similar metrics come into play. To determine the *quantitative growth scale*, we need to approach our microservices with the qualitative growth scale in mind: the quantitative growth scale is defined by translating the qualitative growth scale into a measurable quantity. For example, if the qualitative growth scale of our microservice is measured in "eyeballs" (e.g., how many people open a phone application), and each "eyeball" results in two requests to our microservice and one database transaction, then our quantitative growth scale is measured in terms of requests and transactions, resulting in requests per second and transactions per second as the two key quantities determining our scalability.

The importance of choosing accurate qualitative and quantitative growth scales cannot be overemphasized. As we will soon see, the growth scale will be used when making predictions about the service's operational costs, hardware needs, and limitations.

Efficient Use of Resources

When considering the scalability of large-scale distributed systems like microservice ecosystems, one of the most useful abstractions we can make is to treat properties of our hardware and infrastructure systems as *resources*. CPU, memory, data storage, and the network are similar to resources in the natural world: they are finite, they are physical objects in the real world, and they must be distributed and shared between various key players in the ecosystem. As we discussed briefly in "Organizational Challenges" on page 20, hardware resources are expensive, valuable, and sometimes rare, which leads to fierce competition for resources within the microservice ecosystem.

The organizational challenge of resource allocation and distribution can be alleviated by giving business-critical microservices a greater share of the resources. Resource needs can be prioritized by categorizing various microservices within the ecosystem according to their importance and value to the overall business: if resources are scarce across the ecosystem, the most business-critical services can be given higher priority with regard to resource allocation.

The technical challenge of resource allocation and distribution presents some difficulty, because many decisions need to be made about the first layer (the hardware layer) of the microservice ecosystem. Microservices can be given dedicated hardware so that only one service will run on each host, but this can be rather expensive and an inefficient use of hardware resources. Many engineering organizations opt to share hardware among multiple microservices, and each host will run several different services—a practice that is, in most cases, a more efficient use of hardware resources.

The Dangers of Shared Hardware Resources

While running many different microservices on one machine (that is, sharing machines between microservices) is usually a more efficient use of hardware resources, care must be taken to ensure that the microservices are sufficiently isolated and don't compromise the performance, efficiency, or availability of their neighboring microservices. Containerization (using Docker) along with resource isolation can help prevent microservices from being harmed by badly behaved neighbors.

One of the most effective ways to allocate and distribute hardware resources across a microservice ecosystem is to fully abstract away the notion of a host and replace it with hardware resources using resource abstraction technologies like Apache Mesos. Using this level of resource abstraction allows resources to be allocated dynamically, eliminating many of the pitfalls associated with resource allocation and distribution in large-scale distributed systems like microservice ecosystems.

Resource Awareness

Before hardware resources can be efficiently allocated and distributed to microservices within the microservice ecosystem, it is important to identify the *resource requirements* and *resource bottlenecks* of each microservice. Resource requirements are the specific resources (CPU, RAM, etc.) that each microservice needs; identifying these is essential for running a scalable service. Resource bottlenecks are the scalability and performance limitations of each individual microservice that are dependent on features of its resources.

Resource Requirements

The *resource requirements* of a microservice are the hardware resources the microservice needs in order to run properly, to process tasks efficiently, and to be scaled vertically and/or horizontally. The two most important and relevant hardware resources tend to be, unsurprisingly, CPU and RAM (in multithreaded environments, threads become the third important resource). Determining the resource requirements of a microservice then entails quantifying the CPU and RAM that *one instance* of the service needs in order to run. This is essential for resource abstraction, for resource allocation and distribution, and for determining the overall scalability and performance of the microservice.

Identifying Additional Resource Requirements

While CPU and RAM are the two most common resource requirements, it's important to keep an eye out for other resources that a microservice may need within the ecosystem. These can be hardware resources like database connections or application platform resources like logging quotas. Being aware of the needs of a specific microservice can do a lot to improve scalability and performance.

Calculating the specific resource requirements of a microservice can be a tricky, lengthy process, because there are many relevant factors. The key here, as I mentioned earlier, is to determine what the requirements are for only *one instance* of the service. The most effective and efficient way to scale our service is to scale it horizontally: if our traffic is about to increase, we want to add a few more hosts and deploy our service to those new hosts. In order for us to know how many hosts we need to add, we need to know what our service looks like running on only one host: how much traffic can it handle? how much CPU does it utilize? how much memory? Those numbers will tell us exactly what the resource requirements of our microservice are.

Resource Bottlenecks

We can discover and quantify the performance and scalability limitations of our microservices by identifying *resource bottlenecks*. A resource bottleneck is anything inherent about the way the microservice utilizes its resources that limits the scalability of the application. This could be an infrastructure bottleneck or something within the architecture of the service that prevents it from being scalable. For example, the number of open database connections a microservice needs can be a bottleneck if it nears the connection limit of the database. Another example of a common resource bottleneck is when microservices need to be vertically scaled (rather than horizontally scaled, where more instances/hardware is added) when they experience an increase in traffic: if the only way to scale a microservice is to increase the resources of each instance (more CPU, more memory), then the two principles of scalability (concurrency and partitioning) are abandoned.

Some resource bottlenecks are easy to identify. If your microservice can only be scaled to meet growing traffic by deploying it to machines with more CPU and memory, then you have a scalability bottleneck and need to refactor the microservice so that it can be scaled horizontally rather than vertically, using concurrency and partitioning as your guiding principles.

The Pitfalls of Vertical Scaling

Vertical scaling isn't a sustainable or scalable way to architect microservices. It may appear to work out all right in situations where each microservice has dedicated hardware, but it will not work well with the new hardware abstraction and isolation technologies that are used in the tech world today, like Docker and Apache Mesos. Always optimize for concurrency and partitioning if you want to build a scalable application.

Other resource bottlenecks are not as obvious, and the best way to discover them is to run extensive load testing on the service. We will cover load testing in much greater detail in ???.

Capacity Planning

One of the most important requirements of building a scalable microservice is ensuring that it will have access to necessary and required hardware resources as it scales. Efficiently using resources, planning for growth, and designing a microservice for perfect efficiency and scalability from the ground up is all quickly made useless if no hardware resources are available when the microservice needs to host more production traffic. This challenge is especially relevant for microservices that are optimized for horizontal scalability.

In addition to the technical challenges that accompany this potential problem, engineering organizations are often faced with larger organizational-level and business-relevant issues that come along for the ride: hardware resources cost quite a bit of money, businesses and individual development teams within them have budgets to adhere to, and these budgets (which tend to include hardware) need to be planned for in advance. To ensure that microservices can scale properly when traffic increases, we can perform scheduled *capacity planning*. The principles of capacity planning are pretty straightforward: determine the hardware needs of each microservice in advance, build the needs into the budget, and make sure that the required hardware is reserved.

To determine the hardware needs of each service, we can use the growth scales (both quantitative and qualitative), key business metrics and traffic predictions, the known resource bottlenecks and requirements, and historical data about the microservice's traffic. This is where qualitative and quantitative growth scales come in especially handy, because they allow us to figure out precisely how the scalability behavior of our microservices relate to high-level business predictions. For example, if we know that (1) our microservice scales with unique visitors to the overall product, (2) each unique visitor corresponds to a certain number of requests per second made to our microservice, and (3) that the company predicts that the product will receive 20,000 new unique visitors in the next quarter, then we'll know exactly what our capacity needs will be for the next quarter.

This needs to be built into the budget of each development team, each engineering organization, and each company. Running this exercise on a scheduled basis *before* budgeting is determined can help engineering organizations make sure that hardware resources are never unavailable simply because resource budgeting wasn't completed or prepared for. The important thing here (from both the engineering and business perspectives) is to recognize the cost of inadequate capacity planning: microservices that can't scale properly because of hardware shortages lead to decreased availability within the entire ecosystem, which leads to outages, which costs the company money.

Lead Time for New Hardware Requests

One potential problem that's commonly overlooked by development teams during the capacity planning phase is that the hardware that is needed for the microservice might not exist at the time of planning and may need to be acquired, installed, and configured before any microservices can run on it. Before scheduling capacity planning, take care to find out the exact lead time needed for acquiring new hardware in order to avoid long shortages in critical times, and allow some room for delays in the process.

Once the hardware resources have been secured and dedicated to each microservice, capacity planning is complete. Determining when and how to allocate the hardware after the planning phase is, of course, up to each engineering organization and their development, infrastructure, and operations teams.

Capacity planning can be a really difficult and manual task. Like most manual tasks within engineering, it introduces new modes of failure: manual calculations can be off, and even a small shortage can prove disastrous to business-critical services. Automating the majority of the capacity planning process away from development and operations teams cuts down on potential errors and failures, and a great way to accomplish this is to build and run a capacity planning self-service tool within the application platform layer of the microservice ecosystem.

Dependency Scaling

The scalability of a microservice's dependencies can present a scalability problem of its own. A microservice that is architected, built, and run to be perfectly scalable in every way still faces scalability challenges if it's dependencies cannot scale with it. If even one critical dependency is unable to scale with its clients, then the entire dependency chain suffers. Ensuring that all dependencies will scale with a microservice's expected growth is essential for building production-ready services.

This challenge is relevant to every individual microservice and every part of the microservice ecosystem stack, which means that microservice teams also need to make sure that their service isn't a scalability bottleneck for its clients. In other words, additional complexity is introduced by the rest of the microservice ecosystem. The inevitable additional traffic and growth from a microservice's clients need to be prepared for.

Qualitative Growth Scales and Dependency Scalability

When dealing with incredibly complex dependency chains, making sure that all microservice teams tie the scalability of their services to high-level business metrics (using the qualitative growth scale) can make sure that all services are properly prepared for expected growth, even when cross-team communication becomes difficult.

The problem of dependency scaling is an especially strong argument for the implementation of scalability and performance standards across every part of the microservice ecosystem. Most microservices do not live in isolation. Nearly every single microservice is a small part of large, intertwined, intricate dependency chains. In most cases, scaling the entire overall product, the organization, and the ecosystem effectively requires that each piece of the system scales together with the rest. Having a small number of super efficient, performant, and scalable microservices in a system

where the rest of the services aren't held to (and don't meet) the same standards renders the efficiency of the standardized services completely moot.

Aside from standardization across the ecosystem, and holding each microservice development team to high scalability standards, it's important that development teams work together across microservice boundaries to ensure that each dependency chain will scale together. The development teams responsible for any dependencies of a microservice need to be alerted when increases in traffic are expected. Cross-team communication and collaboration are essential here: regularly communicating with clients and dependencies about a service's scalability requirements, status, and any bottlenecks can help to guarantee that any services that rely on each other are prepared for growth and aware of any potential scalability bottlenecks. A strategy that I've used to help teams accomplish this is by holding architecture and scalability overview meetings with teams whose services rely on one another. In these meetings, we cover the architecture of each service and its scalability limitations, then discuss together what needs to be done to scale the entire set of services.

Traffic Management

As services scale, and the number of requests each service must handle grows, a scalable, performant service must also handle traffic intelligently. There are several aspects to managing traffic in a scalable, performant way: first of all, the growth scale (quantitative and qualitative) needs to be used to predict future increases (or decreases) in traffic; second, the traffic patterns must be well understood and prepared for; and third, microservices need to be able to intelligently handle increases in traffic, as well as surges or bursts of traffic.

We've already covered the first aspect earlier in this chapter: understanding the growth scales (both quantitative and qualitative) of a microservice allows us to understand current traffic loads on the service as well as prepare for future traffic growth.

Understanding current traffic patterns helps when interacting with the service on the ground floor in a lot of really interesting ways. When traffic patterns are clearly identified, both in terms of the requests per second sent to the service over time and all key metrics (see ???, for more about key metrics), changes to the service, operational downtimes, and deployments can be scheduled to avoid peak traffic times, cutting down on possible future outages if a bug is deployed and on potential downtime if the microservice is restarted while experiencing peak traffic load. Closely monitoring the traffic in light of the traffic patterns and tuning the monitoring thresholds carefully with the traffic patterns of the microservice in mind can help catch any issues and incidents quickly before they cause an outage or lead to decreased availability (the principles of production-ready monitoring are covered in greater detail in ???).

When we can predict future traffic growth and understand the current and past traffic patterns well enough to know how the patterns will change with expected growth, we can perform load testing on our services to make sure that they behave as we expect under heavier traffic loads. The details of load testing are covered in ???.

The third aspect of traffic management is where things get especially tricky. The way a microservice handles traffic should be scalable, which means it should be prepared for drastic changes in traffic, especially bursts of traffic, handle them carefully, and prevent them from taking down the service entirely. It's easier said than done, because even the most well-monitored, scalable, and performant microservices can experience monitoring, logging, and other general issues if traffic suddenly spikes. These sorts of spikes should be prepared for at the infrastructure level, within all monitoring and logging systems, and by the development team as part of the service's resiliency testing suite.

There's one additional aspect I want to mention that's related to management of traffic between and across various locations. Many microservice ecosystems won't be deployed only in one location, one datacenter, or one city, but rather across multiple datacenters across the country (or even the world). It's not uncommon for datacenters themselves to experience large-scale outages, and when this happens, the entire microservice ecosystem can (and usually will) go down with the datacenter. Distributing and routing traffic appropriately between datacenters is the responsibility of the infrastructure level (in particular, the communication layer) of the microservice ecosystem, but each microservice needs to be prepared to re-route traffic from one datacenter to another without the service experiencing any decreased availability.

Task Handling and Processing

Every microservice in the microservice ecosystem will need to process tasks of some sort. That is, every microservice will be receiving requests from upstream client services who either need some sort of information from the microservice or need the microservice to compute or process something and then return information about that computation or process, and then the microservice will need to fulfill that request (usually by communicating with downstream services in addition to doing some work of its own) and return any requested information or response to the client that sent the request.

Programming Language Limitations

Microservices can accomplish this and play their required role in a myriad of ways, and the ways in which they will perform computations, interact with downstream services, and process various tasks will depend on the language that the service is written in, and consequently, on the architecture of the service (which is, in many ways, determined by the language). For example, a microservice written in Python

has a number of ways that it can process various tasks, some of which require the use of asynchronous frameworks (like Tornado) and others which can utilize messaging technologies like RabbitMQ and Celery to efficiently process tasks. For these reasons, a microservice's ability to handle and process tasks in a scalable and performant manner is dictated in part by choice of language.

Beware of Scalability and Performance Limitations of Programming Languages

Many programming languages are not optimized for the performance and scalability requirements of microservice architecture, or do not have scalable or performant frameworks that allow microservices to process tasks efficiently.

Because of the limitations introduced by language choice when it comes to a microservice's ability to process tasks efficiently, language choice becomes extremely important in microservice architecture. To many developers, one of the selling points of the adoption of microservice architecture is the ability to write a microservice in any language, and this is usually true, but with a caveat: programming language constraints need to be taken into account, and language choice should be determined not by whether a language is fashionable or fun (or even whether it is the most common language that the development team is familiar with), but with the performance and scalability limitations of each potential language held as the deciding factors. There is no one "best" language to write a microservice in, but there *are* languages that are better suited than others to certain types of microservices.

Handling Requests and Processing Tasks Efficiently

Language choice aside, production-readiness standardization requires each microservice to be both scalable and performant, which means that microservices need to be able to handle and process a large number of tasks at the same time, handle and process those tasks efficiently, and be prepared for tasks and requests to increase in the future. With this in mind, development teams should be able to answer three basic questions about their microservices: how their microservice processes tasks, how efficiently their microservice processes those tasks, and how their microservice will perform as the number of requests scales.

To ensure scalability and performance, microservices need to process tasks efficiently. In order to do this, they need to have both concurrency and partitioning. Concurrency requires that the service can't have one single process that does all of the work: that process will pick up one task at a time, complete the steps in a specific order, and then move on to the next, which is a relatively inefficient way to process tasks. Instead of architecting our service to use a single process, we can introduce concurrency so that each task is broken up into smaller pieces.

Write Microservices in Programming Languages That Are Optimized for Concurrency and Partitioning

Some languages are better suited for efficient (concurrent and partitioned) task handling and processing than others. When writing a new microservice, make sure that the language the service is being written in won't introduce scalability and performance constraints on the microservices. Microservices that are already written in languages with efficiency limitations can (and should) be rewritten in more appropriate languages, a time consuming but incredibly rewarding task that can drastically improve scalability and performance. For example, if you are optimizing for concurrency and partitioning, and want to use an asynchronous framework to help you accomplish that, writing your service in Python (rather than C++, Java, or Go—three languages built for concurrency and partitioning) is going to introduce a lot of scalability and performance bottlenecks that will be difficult to mitigate.

Taking the smaller pieces of these tasks, we can process them more efficiently using partitioning, where each task is not only broken up into small pieces but can be processed in parallel. If we have a large number of small tasks, we can process then all at the same time by sending them to a set of workers that can process them in parallel. If we need to process more tasks, we can easily scale with the increased demand by adding additional workers to process the new tasks without affecting the efficiency of our system. Together, concurrency and partitioning help ensure that our microservice is optimized for both scalability and partitioning.

Scalable Data Storage

Microservices need to *handle data in a scalable and performant way*. The way in which a microservice stores and handles data can easily become the most significant limitation or constraint that keeps it from becoming scalable and performant: choosing the wrong database, the wrong schema, or a database that doesn't support test tenancy can end up compromising the overall availability of a microservice. Choosing the right database for a microservice is a topic that, like all the other topics covered in this book, is incredibly complex, and we will only scratch the surface in this chapter. In the following sections, we'll take a look at several things to consider when choosing databases in microservice ecosystems, and then at some database challenges that are specific to microservice architecture.

Database Choice in Microservice Ecosystems

Building, running, and maintaining databases in large microservice ecosystems is not an easy task. Some companies adopting microservice architecture opt to allow devel-

opment teams to choose, build, and maintain their own databases, while others will decide on at least one database option that works for the majority of the microservices at the company, and build a separate team to run and maintain the database(s) so that developers can focus solely on their own microservices.

If we think about microservice architecture as being composed of four separate layers (see "Microservice Architecture" on page 9 for more details) and recognize that, thanks to the Inverse Conway's Law, the engineering organizations of companies that adopt microservice architecture will mirror the architecture of its product, then we can see where the responsibility for choosing the appropriate databases, building them, running them, and maintaining them lies: either in the application platform layer, which would allow databases to be provided as a service to microservice teams, or the microservice layer, where the database used by a microservice is considered part of the service. I've seen both of these setups in practice at various companies, and some work better than others. I've also noticed that one approach to this works particularly well: offering databases as a service within the application platform layer, and then allowing individual microservice development teams to run their own database if the databases offered as part of the application platform do not fit their specific needs.

The most common types of databases are *relational databases* (SQL, MySQL) and *NoSQL databases* (Cassandra, Vertica, MongoDB, and key-value stores like Dynamo, Redis, and Riak). Choosing between a relational database and a NoSQL database, and then choosing the specific appropriate database for a microservice's needs depends on the answers to several questions:

- What are the needed transactions per second of each microservice?
- What type of data does each microservice need to store?
- What is the schema needed by each microservice? And how often will it need to be changed?
- Do the microservices need strong consistency or eventual consistency?
- Are the microservices read-heavy, write-heavy, or both?
- Does the database need to be scaled horizontally or vertically?

Regardless of whether the database is maintained as part of the application platform or by each individual microservice development team, database choice should be driven by the answers to those questions. For example, if the database in question needs to be scaled horizontally, or if reads and writes need to be made in parallel, then a NoSQL database should be chosen, since relational databases struggle with horizontal scaling and parallel reads and writes.

Database Challenges in Microservice Architecture

There are several challenges with databases that are specific to microservice architecture. When databases are shared among microservices, competition for resources kicks in, and some microservices may utilize more than their fair share of the available storage. Engineers building and maintaining shared databases need to design their data storage solutions so that the database can be easily scaled if any of the tenant microservices either require additional space or are running the risk of using up all available space.

Watch Out for Database Connections

Some databases have strict limitations on the number of database connections that can be open simultaneously. Make sure that all connections are closed appropriately to avoid compromising both a service's availability and the availability of the database to all microservices that use it.

Another challenge microservices often face, especially once they've built and standardized stable and reliable development cycles and deployment pipelines, is the handling of test data from end-to-end testing, load testing, and any test writes done in staging. As mentioned in "The Deployment Pipeline" on page 28, the staging phase of the deployment pipeline requires reading and/or writing to databases. If full staging has been implemented, then the staging phase will have its own separate test and staging database, but partial staging requires read and write access to production servers, so great care needs to be taken to ensure that test data is handled appropriately: it needs to be clearly marked as test data (a process known as *test tenancy*), and then all test data must be deleted at regular intervals.

Evaluate Your Microservice

Now that you have a better understanding of scalability and performance, use the following list of questions to assess the production-readiness of your microservice(s) and microservice ecosystem. The questions are organized by topic, and correspond to the sections within this chapter.

Knowing the Growth Scale

- What is this microservice's qualitative growth scale?
- What is this microservice's quantitative growth scale?

Efficient Use of Resources

- Is the microservice running on dedicated or shared hardware?
- Are any resource abstraction and allocation technologies being used?

Resource Awareness

- What are the microservice's resource requirements (CPU, RAM, etc.)?
- How much traffic can one instance of the microservice handle?
- How much CPU does one instance of the microservice require?
- How much memory does one instance of the microservice require?
- Are there any other resource requirements that are specific to this microservice?
- What are the resource bottlenecks of this microservice?
- Does this microservice need to be scaled vertically, horizontally, or both?

Capacity Planning

- Is capacity planning performed on a scheduled basis?
- What is the lead time for new hardware?
- How often are hardware requests made?
- Are any microservices given priority when hardware requests are made?
- Is capacity planning automated, or is it manual?

Dependency Scaling

- What are this microservice's dependencies?
- Are the dependencies scalable and performant?
- Will the dependencies scale with this microservice's expected growth?

- Are dependency owners prepared for this microservice's expected growth?

Traffic Management

- Are the microservice's traffic patterns well understood?
- Are changes to the service scheduled around traffic patterns?
- Are drastic changes in traffic patterns (especially bursts of traffic) handled carefully and appropriately?
- Can traffic be automatically routed to other datacenters in case of failure?

Task Handling and Processing

- Is the microservice written in a programming language that will allow the service to be scalable and performant?
- Are there any scalability or performance limitations in the way the microservice handles requests?
- Are there any scalability or performance limitations in the way the microservice processes tasks?
- Do developers on the microservice team understand how their service processes tasks, how efficiently it processes those tasks, and how the service will perform as the number of tasks and requests increases?

Scalable Data Storage

- Does this microservice handle data in a scalable and performant way?
- What type of data does this microservice need to store?
- What is the schema needed for its data?
- How many transactions are needed and/or made per second?
- Does this microservice need higher read or write performance?
- Is it read-heavy, write-heavy, or both?
- Is this service's database scaled horizontally or vertically? Is it replicated or partitioned?
- Is this microservice using a dedicated or shared database?
- How does the service handle and/or store test data?

CHAPTER 7

Documentation and Understanding

A production-ready microservice is documented and understood. Documentation and organizational understanding increase developer velocity while mitigating two of the most significant trade-offs that come with the adoption of microservice architecture: organizational sprawl and technical debt. This chapter explores the essential elements of documenting and understanding a microservice, including how to build comprehensive and useful documentation, how to increase microservice understanding at every level of the microservice ecosystem, and how to implement production-readiness throughout an engineering organization.

Principles of Microservice Documentation and Understanding

I'm going to open this final chapter on the last principle of microservice standardization with a famous story from Russian literature. While it may seem rather unorthodox to quote Dostoyevsky in a book on software architecture, the character Grushenka in *The Brothers Karamazov* captures so perfectly what I believe to be the key of microservice documentation and understanding: "Just know one thing, Rakitka, I may be wicked, but still I gave an onion."

My favorite part of Dostoyevsky's brilliant novel is a tale told by the character Grushenka about an old woman and an onion. The tale goes something like this: there was once an old, bitter woman who was very selfish and heartless. One day, she happened upon a beggar, and for some reason, felt a great deal of pity. She wanted to give something to the beggar, but all she had was an onion, so she gave her onion to the beggar. The old woman eventually died, and thanks to her bitterness and coldness of heart, ended up in hell. After she had suffered for quite some time, an angel came to save her, for God had remembered her one selfless deed in life, and wanted to extend the

same kindness in return. The angel reached out to her with an onion in his hand. The old woman grabbed the onion, but to her dismay, the other sinners around her reached for the onion too. Her cold, bitter nature kicked in, and she tried to fight them off, not wanting any of them to have any piece of the onion, and sadly, as she tried to claw the onion away from them, the onion split into many layers and she and the other sinners fell back into hell.

It's not the most heartwarming tale, but there's a moral to Grushenka's story that I have found remarkably applicable to the practice of microservice documentation: always give an onion.

The importance of thorough, updated documentation for every microservice cannot be emphasized enough. Ask developers working in a microservice ecosystem what their main concerns are, and they'll rattle off a list of features still to be implemented, bugs to be fixed, dependencies that are causing trouble, and things that they don't understand about their own service and the dependencies they rely on. When asked to go into greater detail about the latter two things, they tend to give similar answers: they don't understand how it works, it's a black box, and the documentation is completely useless.

Poor documentation of dependencies and internal tools slows developers down and affects their ability to make their own services production-ready. It prevents them from using dependencies and internal tools correctly and wastes countless engineering hours, because sometimes the only way to figure out what a service or tool does (without proper documentation) is to reverse-engineer it until you understand how it works.

Poor documentation of a service also hurts the productivity of the developers who are contributing to it. For example, the lack of runbooks for an on-call shift means whoever is on call will need to figure out each problem from square one every single time. Without an onboarding guide, each new developer working on the service will need to start from scratch to understand how the service works. Single points of failure and problems with the service will go unnoticed until they cause an outage. New features added to the service will often miss the big picture of how the service actually works.

The goal of good, production-ready documentation is to create and maintain a centralized repository of knowledge about the service. Sharing that knowledge has two components: the bare facts about the service, and organizational understanding of what the service does and where it fits into the organization as a whole. The problem of poor documentation can then be divided into two subproblems: lack of documentation (the facts) and lack of understanding. Solving these two subproblems requires standardizing documentation for every microservice and putting organizational structures into place for sharing microservice understanding.

Grushenka's tale is the golden rule of microservice documentation: always give an onion. Give an onion for your sake, for the sake of fellow developers working on your service, and for the sake of the developers whose services depend on yours.

A Production-Ready Service Is Documented and Understood

- It has comprehensive documentation.
- Its documentation is updated regularly.
- Its documentation contains a description of the microservice; an architecture diagram; contact and on-call information; links to important information; an onboarding and development guide; information about the service's request flow(s), endpoints, and dependencies; an on-call runbook; and answers to frequently asked questions.
- It is well understood at the developer, team, and organizational levels.
- It is held to a set of production-readiness standards and meets the associated requirements.
- Its architecture is reviewed and audited frequently.

Microservice Documentation

The documentation for all microservices in an engineering organization should be stored in a centralized, shared, and easily accessible place. Any developer on any team should be able to find the documentation for every microservice without any difficulty. An internal website containing the documentation for all microservices and internal tools tends to be the best medium for this.

READMEs and Code Comments Are Not Documentation

Many developers limit the documentation of their microservices to a README file in their repository or to comments scattered throughout the code. While having a README is essential, and all microservice code should contain appropriate comments, this is *not* production-ready documentation and requires that developers check out and search through the code. Proper documentation is stored in a centralized place (like a website) where the documentation for all microservices in the engineering organization lives.

The documentation should be updated regularly. Any time a significant change is made to the service, the documentation should be updated. For example, if a new API endpoint is added, information about the endpoint must be added to the documenta-

tion as well. If a new alert is added, then step-by-step instructions on how to triage, mitigate, and resolve the alert should also be added to the service's on-call runbook. If a new dependency is added, then information about that dependency should be added to the documentation. Always give an onion.

The best way to accomplish this is to make the process of updating documentation part of the development workflow. If updating documentation is seen as a separate task aside from (and secondary to) development, then it will never get done and will become part of the technical debt of the service. To reduce technical debt, developers should be encouraged (or, if need be, required) to accompany every significant code change with an update to the documentation.

Make Updating Documentation Part of the Development Cycle

If updating and improving documentation is viewed as secondary to writing code, it will often be pushed off and become part of the technical debt of the service. To avoid this, make documentation updates and improvements a required part of the development cycle of the service.

Documentation should be both comprehensive and useful. It should contain all of the relevant and important facts about the service. After reading through the documentation, a developer should know how to develop and contribute to the service; the architecture of the service; the contact and on-call information for the service; how the service works (request flows, endpoints, dependencies, etc.); how to triage, mitigate, and fix incidents and outages as well as resolve alerts generated by the service; and answers to frequently asked questions about the service.

Most importantly, documentation should be written clearly and should be easy to understand. Jargon-heavy documentation is useless, documentation that is overly technical and doesn't explain things that may be unique to the service is also useless, as is documentation that doesn't go into any significant detail at all. The goal in writing good, clean, and clear documentation is to write it so that it can be understood by any developer, manager, product manager, or executive within the company.

Let's dive a little bit deeper into each of the elements of production-ready microservice documentation.

Description

Each microservice's documentation should begin with a *description* of the service. It should be short, sweet, and to the point. For example, if there is a microservice called *receipt-sender* whose purpose is to send a receipt after a customer completes an order, the description should read:

Description:

After a customer places an order, receipt-sender sends a receipt to the customer via email.

This is essential because it ensures that anyone who finds the documentation will know what role the microservice plays in the microservice ecosystem.

Architecture Diagram

The description of the service should be followed by an *architecture diagram*. This diagram should detail the architecture of the service, including its components, its endpoints, the request flow, its dependencies (both upstream and downstream), and information about any databases or caches. See an example architecture diagram in Figure 7-1.

Architecture diagrams are essential for several reasons. It's nearly impossible to understand how and why a microservice works just by reading through the code, and so a well-designed architecture diagram is an easily understandable visual description and summary of the microservice. These diagrams also aid developers in adding new features by abstracting away the inner workings of the service so that developers can see where and how new features will (or will not) fit. Most importantly, they illuminate issues and problems with the service that would go unnoticed without a complete visual representation of its architecture: it's difficult to discover a service's points of failure by combing through lines of code, but they tend to stick out like sore thumbs in an accurate architecture diagram.

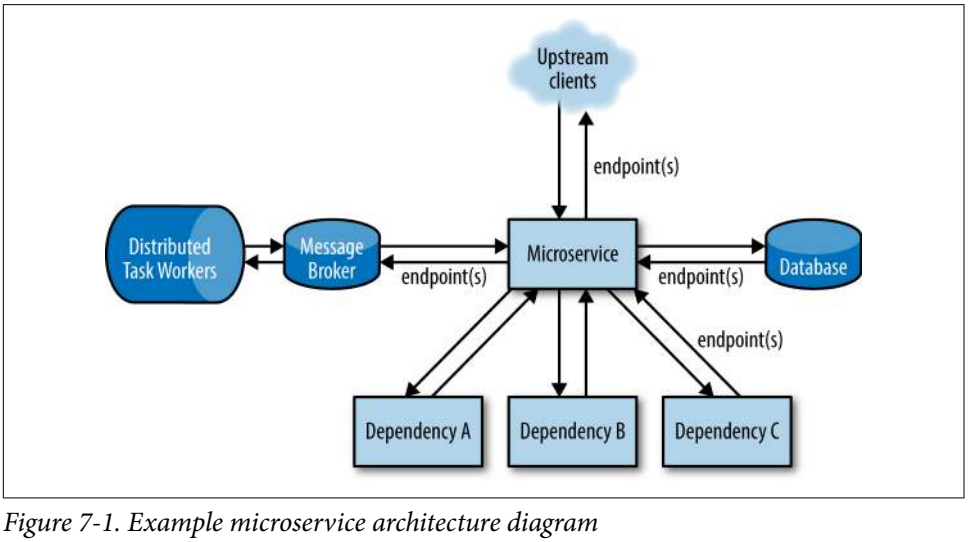

Figure 7-1. Example microservice architecture diagram

Contact and On-Call Information

Chances are, anyone looking at a service's documentation will either be someone on the service team, or someone on a different team who is experiencing trouble with the service or wants to know how the service works. For developers in the second group, having access to information about the team is both useful and necessary, and so several important facts should be included in a *contact and on-call information* section within the documentation.

This section should include the names, positions, and contact information of everyone on the team (including individual contributors, managers, and program/product managers). This makes it easy for developers on other teams to quickly determine who they should contact if they experience a problem with the service or have a question about it. This information is useful, for example, when a developer is experiencing problems with one of their dependencies: knowing who to contact and what their role is on the team makes cross-team communication easy and efficient.

Adding information about the on-call rotation (and keeping it updated so that it reflects who is on call for the service at any given time) will ensure that people will know exactly who to contact for general problems or emergencies: the engineer who is on call for the service.

Links

Documentation needs to be a centralized resource for all the information about a microservice. In order for this to be true, the documentation needs to contain links to the repository (so that developers can easily check out the code), a link to the dashboard, a link to the original RFC for the microservice, and a link to the most recent architecture review slides. Any extra information about other microservices, technologies used by the microservice, etc., that may be useful to the developer should be included in a *links* section of the documentation.

Onboarding and Development Guide

The purpose of an *onboarding and development* section is to make it easy for a new developer to onboard to the team, begin contributing code, add features to the microservice, and introduce new changes into the deployment pipeline.

The first part of this section should be a step-by-step guide to setting up the service. It should walk a developer through checking out the code, setting up the environment, starting the service, and verifying that the service is working correctly (including all commands or scripts that need to be run in order to accomplish this).

The second part should guide the developer through the development cycle and deployment pipeline of the service (details of a production-ready development cycle

and deployment pipeline can be found in "The Development Cycle" on page 26 and "The Deployment Pipeline" on page 28). This should include the technical details (e.g., commands that must be run, along with several examples) of each of the steps: how to check out the code, how to make a change to the code, how to write a unit test for the change (if necessary), how to run the required tests, how to commit their changes, how to send changes for code review, how to make sure that the service is built and released correctly, and then how to deploy (as well as how the deployment pipeline is set up for the service).

Request Flows, Endpoints, and Dependencies

The documentation should also contain critical information about *request flows, endpoints, and dependencies* of the microservice.

Request flow documentation can consist of a diagram of the request flows of the application. This can be the architecture diagram, if the request flow is detailed appropriately within the architecture diagram. Any diagram should be accompanied by a qualitative description of the types of requests that are made to the microservice and how they are handled.

This is also the place to document all API endpoints of the service. A bulleted list of the endpoints with their names and a qualitative description of each along with their responses is usually sufficient. It must be clear and understandable enough that another developer working on a different team could read the descriptions of your service's API endpoints and treat your microservice as a black box, hitting the endpoints successfully and receiving the expected responses.

The third element of this section is information about the service's dependencies. List the dependencies, the relevant endpoints of these dependencies, and any requests the service makes to them, along with information about their SLAs, any alternatives/caching/backups in place in case of failure, and links to their documentation and dashboards.

On-Call Runbooks

As covered in ???, every single alert should be included in an on-call runbook and accompanied by step-by-step instructions describing how it should be triaged, mitigated, and resolved. The on-call runbook should be kept in the centralized documentation of the service, in an *on-call runbook* section, along with both general and detailed guidance on troubleshooting and debugging new errors.

A good runbook will begin with any general on-call requirements and procedures, and then contain a complete list of the service's alerts. For each alert, the on-call runbook should include the alert name, a description of the alert, a description of the problem, and a step-by-step guide on how to triage the alert, mitigate it, and then

resolve it. It will also describe any organizational implications of the alert: the severity of the problem, whether or not the alert signifies an outage, and information about how to communicate any incidents and outages to the team, and if necessary, to the rest of the engineering organization.

Write On-Call Runbooks That Sleepy Developers Can Understand at 2 A.M.

Developers on call for a service may (or, more realistically, will) be paged at any hour of the day, including late at night or very early in the morning. Write your on-call runbooks so that a half-asleep developer will be able to follow along without any difficulty.

Writing good, clear, easily understandable on-call runbooks is extremely important. They should be written so that any developer who is on call for the service or who is experiencing trouble with the service will be able to act quickly, diagnose the problem, mitigate the incident, and resolve, all in an extremely small amount of time in order to keep the downtime of the service very, very low.

Not every alert will be easily mitigated or resolved, and most outages (aside from those caused by code bugs introduced by a recent deployment) haven't been seen before. To equip developers to handle these problems wisely, add a general *troubleshooting and debugging* section to the on-call runbook in the documentation that is filled with tips on how to approach new problems in a strategic and methodical way.

FAQ

An often forgotten element of documentation is a section devoted to answering common questions about the service. Having a "Frequently Asked Questions" section takes the burden of answering common questions off of whomever is on call and, consequently, the rest of the team.

There are two categories of questions that should be answered here. The first are questions that developers on other teams ask about the service. The way to approach answering these questions in an FAQ setting is simple: if someone asks you a question, and you think it might be asked again, add it to the FAQ. The second category of questions are those that come from team members, and the same approach can be taken here: if there's a question about how or why or when to do something related to the service, add it to the FAQ.

Summary: Elements of Production-Ready Microservice Documentation

Production-ready microservice documentation includes:

- A description of the microservice and its place in the overall microservice ecosystem and the business
- An architecture diagram detailing the architecture of the service and its clients and dependencies at a high level of abstraction
- Contact and on-call information about the microservice's development team
- Links to the repository, dashboard(s), the RFC for the service, architecture reviews, and any other relevant or useful information
- An onboarding and development guide containing details about the development process, the deployment pipeline, and any other information that will be useful to developers who contribute code to the service
- Detailed information about the microservice's request flows, SLA, production-readiness status, API endpoints, important clients, and dependencies
- An on-call runbook containing general incident and outage response procedures, step-by-step instructions on how to triage, mitigate, and resolve each alert, and a general troubleshooting and debugging section
- A "Frequently Asked Questions" (FAQ) section

Microservice Understanding

Centralized, updated, and thorough documentation is only one part of production-ready microservice documentation and understanding. Aside from writing and updating documentation, organizational processes should be put into place to ensure that microservices are well understood not only by the individual development teams but by the organization as a whole. In many ways, a well-understood microservice is one that meets every production-readiness requirement.

Microservice understanding is truly indispensable to the developer, the team, and the organization. While the notion of "understanding" a microservice may seem too vague to be useful at first glance, the concept of a production-ready microservice can be used to guide and define microservice understanding at every level. Armed with production-readiness standards and requirements, along with a realistic understanding of organizational complexity and the challenges that microservice architecture brings to the arena, developers can quantify their understanding of each microservice

and (as I've urged the reader earlier in this chapter) can give an onion to the rest of the organization.

For the individual developer, this translates to being able to answer questions about her microservice. For example, when asked if her microservice is scalable, she will be able to look at a list of scalability requirements and confidently answer "Yes," "No," or something in between (e.g., "It meets requirements *x* and *z*, but *y* has not yet been implemented"). Likewise, when asked if her microservice is fault tolerant, she'll be able to rattle off all failure scenarios and possible catastrophes, then explain in detail how she has prepared for these using various types of resiliency testing.

At the team level, *understanding* signifies that the team is aware of where their microservice stands with regard to production-readiness and what needs to be accomplished to bring their service to a production-ready state. This has to be a cultural element of each team in order for it to be successful: production-readiness standards and requirements need to drive the decisions made by the team and be seen not merely as boxes to check off on a checklist, but rather as principles that guide the team toward building the best possible microservice.

Understanding needs to be built into the fabric of the organization itself. This requires that production-readiness standards and requirements become part of the organizational process. Before a service is even built, and a *request for comments* (RFC) is sent around for review, the service can be evaluated against the production-readiness standards and requirements. Developers, architects, and operations engineers can make sure that the service is built for stability, reliability, scalability, performance, fault tolerance, catastrophe-preparedness, proper monitoring, and appropriately documented and understood before it even begins running—ensuring that once the new service begins to host production traffic, it has been architected and optimized for availability and can be trusted with production traffic.

It's not enough to only review and architect for production-readiness at the beginning of a microservice's lifecycle. Existing services need to be reviewed and audited constantly so that the quality of each microservice is kept at a sufficiently high level, ensuring high availability and trust across various microservice teams and the entire microservice ecosystem. Automating these production-readiness audits of existing services and internally publicizing the results can help to establish awareness across the organization about the quality of the overall microservice ecosystem.

Architecture Reviews

One thing I've learned after driving these production-readiness standards and their requirements across over a thousand different microservices and their development teams is that the most immediately effective way to accomplish microservice understanding is to hold scheduled *architecture reviews* for each microservice. A good architecture review is a meeting where any and all developers and site reliability engi-

neers (or other operations engineers) working on the service meet in a room, draw up the architecture of the service on a whiteboard, and thoroughly evaluate its architecture.

Within several minutes into this exercise, it tends to become very clear precisely what the scope of understanding is at the developer and team levels. Talking through the architecture, developers will quickly discover scalability and performance bottlenecks, previously undiscovered points of failure, possible outages and future incidents and failures and catastrophe scenarios, and new features that should be added. Poor architectural decisions that were made in the past will become obvious, and old technologies that should be replaced by newer and/or better ones will stand out. To ensure that evaluation and discussion is productive and objective, it's helpful to bring in developers from other teams (especially those in infrastructure, DevOps, or site reliability engineering) who have experience in large-scale distributed systems architecture (and the organization's specific microservice ecosystem) and will be able to point out problems that developers may not notice.

Each meeting should produce a new, updated architecture diagram for the service, along with a list of projects to tackle in the coming weeks and months. The new diagram should definitely be added to the documentation, and projects can be included in each service's *roadmap* (see "Production-Readiness Roadmaps" on page 72) and *objectives and key results* (OKRs).

Because microservice development moves rather quickly, microservices evolve at a rapid pace and the lower layers of the microservice ecosystem will be constantly changing. In order to keep the architecture and its understanding relevant and productive, these meetings should be held regularly. I've found that a good rule of thumb is to schedule them so that they align with OKR and project planning. If projects and OKRs are planned and scheduled quarterly, then quarterly architecture reviews should be held each quarter before the planning cycle begins.

Production-Readiness Audits

To make sure that a microservice meets production-readiness standards and requirements and is actually production-ready, the team can run a *production-readiness audit* on the service. Running an audit is quite simple: the team sits down with a checklist of the production-readiness requirements and checks off whether or not their service meets each requirement. This enables understanding of a service: each developer and team will know, by the end of the audit, exactly where their service stands and where things can be improved.

The structure of an audit should mirror the production-readiness standards and requirements that the engineering organization has adopted. The team should use the audits to quantify the stability, reliability, scalability, fault tolerance, catastrophe-preparedness, performance, monitoring, and documentation of the service. As I've

described in earlier chapters, each of these standards is accompanied by a set of requirements that can be used to bring each service up to those standards—developers can adjust these requirements of each production-readiness standard so that they meet the needs and goals of the organization. The exact requirements will depend on the details of the company's microservice ecosystem, but the standards and their basic components are relevant across every ecosystem (see Appendix A for a summary checklist containing the production-readiness standards and their general requirements).

Production-Readiness Roadmaps

Once a microservice development team has completed a thorough production-readiness audit of their microservice and the team understands whether their service is production-ready, the next step is to plan how to bring the service to a production-ready state. Audits make this easy: at this point, the team has a checklist of which production-readiness requirements their service doesn't meet, and all that is left to do is to satisfy each unfulfilled requirement.

This is where *production-readiness roadmaps* can be developed, and I've found them to be an extremely useful piece of the production-readiness and microservice understanding process. Each microservice is different, and the implementation details of each unsatisfied requirement will vary between services, so producing a detailed roadmap that documents all of the implementation details will guide the team toward making their microservice production-ready. Requirements that need to be met can be accompanied by the technical details, problems that have arisen (outages and incidents) that are related to the requirement, a link to some ticket in a task-management system, and the name(s) of the developer(s) who will be working on the project.

The roadmap and the list of unsatisfied production-readiness requirements it contains can become part of whatever planning and (if used at the company) OKRs are in store for the service. Satisfying production-readiness requirements works best when the process goes hand in hand both with feature development and with the adoption of new technologies. Making each service in the microservice ecosystem stable, reliable, scalable, performant, fault tolerant, catastrophe-prepared, monitored, documented, and understood is a straightforward, quantifiable way to guarantee that each service is truly production-ready, ensuring the availability of the entire microservice ecosystem.

Production-Readiness Automation

Architecture reviews, audits, and roadmaps solve the challenges of microservice understanding at the developer and team levels, but understanding at an organizational level requires an additional component. As I've presented it so far, all of the work that goes into building a production-ready microservice is mostly manual,

requiring developers to individually follow each audit step, make tasks and lists and roadmaps and check off individual requirement boxes. Manual work like this often gets put on the back burner to join the rest of the technical debt, even in the most productive and production-readiness driven teams.

One of the key principles of software engineering in practice is this: if you have to do something manually more than once, automate it so that you never have to do it again. This applies to operational work, it applies to any one-off, ad hoc situations and anything you need to type into a terminal, and not surprisingly, it applies to enforcing production-readiness standards across an engineering organization. Automation is the best onion you can give to your development teams.

It's easy to make a list of the production-readiness requirements for every microservice. I've done it myself at Uber, I've seen other developers implement the very same production-readiness standards in this book at their own companies, and I've created a template checklist (Appendix A, *Production-Readiness Checklist*) that you, the reader, can use. A list like this makes automating the checklist rather easy. For example, to check for fault tolerance and catastrophe-preparedness, you can run automated checks to ensure that the proper resiliency tests are in place, are running, and that each microservice passes the tests with flying colors.

The difficulty in automating each of these production-readiness checks will depend entirely on the complexity of your internal services within each layer of the microservice ecosystem. If all microservices and self-service tools have decent APIs, automation is a breeze. If your services have trouble communicating, or if any self-service internal tools are finicky or poorly written, you're going to have a bad time (and not just with production-readiness, but with the integrity of your service and the entire microservice ecosystem).

Automating production-readiness increases organizational understanding in several extremely important and effective ways. If you automate these checks and run them constantly, teams in the organization will always know where each microservice stands. Publicize these results internally, give each microservice a production-readiness score measuring how production-ready their service is, require business-critical services to have a high minimum production-readiness score, and gate deployments. Production-readiness can be made part of the engineering culture, and this is one surefire way you can accomplish that.

Evaluate Your Microservice

Now that you have a better understanding of documentation, use the following list of questions to assess the production-readiness of your microservice(s) and microservice ecosystem. The questions are organized by topic, and correspond to the sections within this chapter.

Microservice Documentation

- Is the documentation for all microservices stored in a centralized, shared, and easily accessible place?
- Is the documentation easily searchable?
- Are significant changes to the microservice accompanied by updates to the microservice's documentation?
- Does the microservice's documentation contain a description of the microservice?
- Does the microservice's documentation contain an architecture diagram?
- Does the microservice's documentation contain contact and on-call information?
- Does the microservice's documentation contain links to important information?
- Does the microservice's documentation contain an onboarding and development guide?
- Does the microservice's documentation contain information about the microservice's request flow, endpoints, and dependencies?
- Does the microservice's documentation contain an on-call runbook?
- Does the microservice's documentation contain an FAQ section?

Microservice Understanding

- Can every developer on the team answer questions about the production-readiness of the microservice?
- Is there a set of principles and standards that all microservices are held to?
- Is there an RFC process in place for every new microservice?
- Are existing microservices reviewed and audited frequently?
- Are architecture reviews held for every microservice team?
- Is there a production-readiness audit process in place?
- Are production-readiness roadmaps used to bring the microservice to a production-ready state?
- Do the production-readiness standards drive the organization's OKRs?
- Is the production-readiness process automated?

APPENDIX A

Production-Readiness Checklist

This will be a checklist to run over all microservices—manually or in an automated way.

A Production-Ready Service Is Stable and Reliable

- It has a standardized development cycle.
- Its code is thoroughly tested through lint, unit, integration, and end-to-end testing.
- Its test, packaging, build, and release process is completely automated.
- It has a standardized deployment pipeline, containing staging, canary, and production phases.
- Its clients are known.
- Its dependencies are known, and there are backups, alternatives, fallbacks, and caching in place in case of failures.
- It has stable and reliable routing and discovery in place.

A Production-Ready Service Is Scalable and Performant

- Its qualitative and quantitative growth scales are known.
- It uses hardware resources efficiently.
- Its resource bottlenecks and requirements have been identified.
- Capacity planning is automated and performed on a scheduled basis.
- Its dependencies will scale with it.

- It will scale with its clients.
- Its traffic patterns are understood.
- Traffic can be re-routed in case of failures.
- It is written in a programming language that allows it to be scalable and performant.
- It handles and processes tasks in a performant manner.
- It handles and stores data in a scalable and performant way.

A Production-Ready Service Is Fault Tolerant and Prepared for Any Catastrophe

- It has no single point of failure.
- All failure scenarios and possible catastrophes have been identified.
- It is tested for resiliency through code testing, load testing, and chaos testing.
- Failure detection and remediation has been automated.
- There are standardized incident and outage procedures in place within the microservice development team and across the organization.

A Production-Ready Service Is Properly Monitored

- Its key metrics are identified and monitored at the host, infrastructure, and microservice levels.
- It has appropriate logging that accurately reflects the past states of the microservice.
- Its dashboards are easy to interpret and contain all key metrics.
- Its alerts are actionable and are defined by signal-providing thresholds.
- There is a dedicated on-call rotation responsible for monitoring and responding to any incidents and outages.
- There is a clear, well-defined, and standardized on-call procedure in place for handling incidents and outages.

A Production-Ready Service Is Documented and Understood

- It has comprehensive documentation.
- Its documentation is updated regularly.
- Its documentation contains a description of the microservice; an architecture diagram; contact and on-call information; links to important information; an onboarding and development guide; information about the service's request flow(s), endpoints, and dependencies; an on-call runbook; and answers to frequently asked questions.
- It is well understood at the developer, team, and organizational levels.
- It is held to a set of production-readiness standards and meets the associated requirements.
- Its architecture is reviewed and audited frequently.

About the Author

Susan Fowler is a site reliability engineer at Uber Technologies, where she splits her time between running a production-readiness initiative across all Uber microservices and embedding within business-critical teams to bring their services to a production-ready state. She worked on application platforms and infrastructure at several small startups before joining Uber, and before that, studied particle physics at Penn, where she searched for supersymmetry and designed hardware for the ATLAS and CMS detectors.

Colophon

The animals on the cover of *Production-Ready Microservices* are leafcutter bees (of the genus *Megachile*). There are over 1,500 species of this insect, which is widespread throughout the world. One species from Indonesia, *Megachile pluto*, is thought to be the largest bee in the world: individuals can be up to 0.9–1.5 inches long.

Leafcutter bees gain their name from the female's common activity of cutting neat semicircles from the edges of leaves. She then carries these disc-shaped leaf pieces to her nest, which can be built in various places such as ready-made hollows, burrows in the ground, or rotting wood that the bee can bore into. Nests are between 4–8 inches long, cylindrical, and lined with leaf pieces in an overlapping pattern. These insects do not live in colonies, though it's possible for individuals to nest near each other.

The females arrange their nests in separate cells (building from the inside out) and lay one egg within each, along with a regurgitated pollen-and-nectar ball for the larva to eat. It is theorized that the leaves keep the larva's food from drying out until it can be eaten.

Adult bees also feed on nectar and pollen, and are very efficient pollinators due to their vigorous swimming-like motion while inside flowers (which shakes a great deal of pollen loose and coats the long hairs on the insect's abdomen). Females often need to take 10–15 trips to provision an individual nest cell, further increasing their effectiveness in cross-pollination. Thus, these bees are welcome inhabitants in many gardens and farms; artificial nesting boxes or tubes can be placed to attract them.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from Lydekker's *Royal Natural History*. The cover fonts are URW Typewriter 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.