O'REILLY®

Cloud Native

Using Containers, Functions, and Data to Build Next-Generation Applications

Boris Scholl,
Trent Swanson
& Peter Jausovec

Cloud Native

Using Containers, Functions, and Data to Build Next-Generation Applications

Boris Scholl, Trent Swanson, and Peter Jausovec

Cloud Native

by Boris Scholl, Trent Swanson, and Peter Jausovec

Printed in the United States of America.

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

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

Acquisitions Editor: Kathleen Carr

Development Editor: Nicole Tache

Production Editor: Elizabeth Kelly

Copyeditor: Octal Publishing, Inc.

Proofreader: Rachel Monaghan

Indexer: Ellen Troutman-Zaig

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

September 2019: First Edition

Revision History for the First Edition

- 2019-08-21: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492053828 for release details.

Preface

Thought leaders across different companies and industries have been restating Watts Humphrey's statement, "Every business will become a software business." He was spot on. Software is taking over the world and is challenging the status quo of existing companies. Netflix has revolutionized how we obtain and consume TV and movies, Uber has transformed the transportation industry, and Airbnb is challenging the hotel industry. A couple of years ago that would have been unthinkable, but software has allowed new companies to venture into all industries and establish new thinking and business models.

The previously mentioned companies are often referred to as "born-in-the-cloud companies," which means that at the basis of their offerings are services running in the cloud. Those services are built in a way that companies can quickly react to market and customer demands, release updates and fixes in a short period of time, use the latest technologies, and take advantage of the improved economics provided by the cloud. Services built in a cloud native way have also allowed companies to rethink their business models and move to new ones, such as subscription-based models. Such services are often referred to as *cloud native applications*.

The success and popularity of cloud native applications have led many enterprises to adopt cloud native architectures, even bringing many of the concepts to on-premises applications.

At the heart of cloud native applications are *containers*, *functions*, and

data. There are many books out there focusing on each of these specific technologies. Cloud native applications use all of these technologies and take advantage of and exploit all of the benefits of the cloud. We, the authors, have seen many customers struggle to piece all of those technologies together to design and develop cloud native applications, so we decided to write a book with the goal to provide the foundational knowledge that enables developers and architects alike to get started with designing cloud native applications.

This book starts by laying down the foundation for the reader to understand the basic principles of distributed computing and how they relate to cloud native applications, as well as providing a closer look at containers and functions. Further, it covers service communication patterns, resiliency, and data patterns as well as providing guidance on when to use what. The book concludes by explaining the DevOps approach, portability considerations, and a collection of best practices that we have seen to be useful in successful cloud native applications.

The book is not a step-by-step implementation guide for building cloud native applications for a specific set of requirements. After reading this book, you should have the understanding and knowledge to help design, build, and operate successful cloud native applications. Tutorials are great for working through very specific needs, but a fundamental understanding of building cloud native applications provides teams with the necessary skills to ship successful cloud native applications.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

O’Reilly Online Learning

> **NOTE**
>
> For almost 40 years, *O’Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/cloud-native-1e*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

We would like to thank Nicole Taché, our editor at O'Reilly, as well as the tech reviewers and beta reviewers for their valuable contributions to the book. In addition, we would like to thank Haishi Bai and Bhushan Nene for their thorough reviews and suggestions to improve the quality of the book.

Boris would like to thank his wife, Christina, and his kids, Marie and Anton, for being so patient and supportive during the time he was working on the book.

Trent would like to thank his wife, Lisa, and his son, Mark, for their support and patience while he was working on this book.

Peter would like to thank his wife, Nives, for her support, encouragement, and understanding while he was working nights and weekends on this book.

Chapter 1. Introduction to Cloud Native

What are cloud native applications? What makes them so appealing that the cloud native model is now considered not only for the cloud, but also for the edge? And, finally, how do you design and develop cloud native applications? These are all questions that will be answered throughout this book. But before we dive into the details on the what, why, and how, we want to provide a brief introduction to the cloud native world and some of the fundamental concepts and assumptions that are building the foundation for modern cloud native applications and environments.

Distributed Systems

One of the biggest hurdles that developers face when they build cloud native applications for the first time is that they must deal with services that are not on the same machine, and they need to deal with patterns that consider a network between the machines. Without even knowing it, they have entered the world of distributed systems. A *distributed system* is a system in which individual computers are connected through a network and appear as a single computer. Being able to distribute computing power across a bunch of machines is a great way to accomplish scalability, reliability, and better economics. For example, most cloud providers are using cheaper commodity hardware and solving common problems such as high availability and reliability through software-based solutions.

Fallacies of Distributed Systems

There are couple of incorrect or unfounded assumptions most developers and architects make when they enter the world of distributed systems. Peter Deutsch, a Fellow at Sun Microsystems, was identifying fallacies of distributed computing back in 1994, at a time when nobody thought about cloud computing. Because cloud native applications are, at their core, distributed systems, these fallacies still have validity today. Following is the list of the fallacies that Deutsch described, with their meanings applied to cloud native applications:

The network is reliable

> Even in the cloud you cannot assume that the network is reliable. Because services are typically placed on different machines, you need to develop your software in a way that it accounts for potential network failures, which we discuss later in this book.

Latency is zero

> Latency and bandwidth are often confused, but it is important to understand the difference. Latency is how much time goes by until data is received, whereas bandwidth indicates how much data can be transferred in a given window of time. Because latency has a big impact on user experience and performance, you should take care to do the following:
>
> - Avoid frequent network calls and introducing chattiness to the network.
> - Design your cloud native application in a way that the data is closest to your client by using caching, content delivery networks (CDNs), and multiregion deployments.
> - Use publication/subscription (pub/sub) mechanisms to be notified that there is new data and store it locally to be immediately available. Chapter 3 covers messaging patterns such as pub/sub in more detail.

There is infinite bandwidth

Nowadays, network bandwidth does not seem to be a big issue, but new technologies and areas such as edge computing open up new scenarios that demand far more bandwidth. For example, it is predicted that a self-driving car will produce around 50 terabytes (TB) of data per day. This volume of data requires you to design your cloudnative application with bandwidth usage in mind. Domain-Driven Design (DDD) and data patterns such as Command Query Responsibility Segregation (CQRS) are very useful under such bandwidth-demanding circumstances. Chapter 4 and Chapter 6 cover how to work with data in cloud native applications in more detail.

The network is secure

Two things are often an afterthought for developers: diagnostics and security. The assumption that networks are secure can be fatal. As a developer or architect, you need to make security a priority of your design; for example, by embracing a defense-in-depth approach.

The topology does not change

Pets versus cattle is a meme that gained popularity with the advent of containers. It means that you do not treat any machine as a known entity (pet) with its own set of properties, such as static IPs and so on. Instead, you treat machines as a member of a herd that has no special attributes. This concept is very important with cloud native applications. Because cloud environments are meant to provide elasticity, machines can be added and removed based on criteria such as resource consumption or requests per second.

There is one administrator

In traditional software development, it was quite common to have one person responsible for the environment, installing and upgrading the application, and so forth. Modern cloud architectures and DevOps methods have shifted the way software is built. A modern cloud native application is a composite of many services that need to work together in concert and that are developed by different teams. This makes it

practically impossible for a single person to know and understand the application in its entirety, not to mention trying to fix a problem. Thus, you need to ensure that you have governance in place that makes it easy to troubleshoot issues. Throughout this book, we introduce you to important concepts such as *release management*, *decoupling*, and *logging and monitoring*. Chapter 5 provides a detailed look at common DevOps practices for cloud native applications.

Transport cost is zero

From a cloud native perspective, there are two ways to look at this one. First, transport happens over a network and network costs are not free with most cloud providers. Most cloud providers, for example, do not charge for data ingress, but do charge for data egress. The second way to look at this fallacy is that the cost for translating any payload into objects is not free. For example, serialization and deserialization are usually fairly expensive operations that you need to consider in addition to the latency of network calls.

The network is homogeneous

This is almost not worth listing given that pretty much every developer and architect understands that there are different protocols that they must consider when building their applications.

As mentioned before, although these fallacies were documented a long time ago, they are still a good reminder of the incorrect assumptions people make when entering the cloud native world. Throughout this book, we teach you patterns and best practices that take all of the fallacies of distributed computing into account.

CAP Theorem

The CAP theorem is often mentioned in combination with distributed systems. The CAP theorem states that any networked shared-data system can have at most two of the following three desirable properties:

- Consistency (C) equivalent to having a single up-to-date copy of the data
- High availability (A) of that data (for updates)
- Tolerance to network partitions (P)

The reality is that you will always have network partitions (remember, "the network is reliable" is one of the fallacies of distributed computing). That leaves you with only two choices—you can optimize either for consistency or high availability. Many NoSQL databases such as Cassandra optimize for availability, whereas SQL-based systems that adhere to the principles of ACID (atomicity, consistency, isolation, and durability) optimize for consistency.

The Twelve-Factor App

In the early days of Infrastructure as a Service (IaaS) and Platform as a Service (PaaS), it quickly became obvious that the cloud required a new way of developing applications. For example, on-premises scaling was often done by scaling vertically, meaning adding more resources to a machine. Scaling in the cloud, on the other hand, is usually done horizontally, meaning adding more machines to distribute the load. This type of scaling requires stateless applications, and this is one of the factors described by the *Twelve-Factor App manifesto*. The Twelve-Factor App methodology can be considered the foundation for cloud native applications and was first introduced by engineers at Heroku, derived from best practices for application development in the cloud. Cloud development has evolved since the introduction of the Twelve-Factor manifesto, but the principles still apply. Following are the 12 factors and their meaning for cloud native applications:

1. Codebase

   One codebase tracked in revision control; many deploys.

   There is only one codebase per application, but it can be deployed into many environments such as Dev, Test, and Prod. In cloud native architecture, this translates directly into one codebase per service or function, each having its own Continuous Integration/Continuous Deployment (CI/CD) flow.

2. Dependencies

   Explicitly declare and isolate dependencies.

   Declaring and isolating dependencies is an important aspect of cloud native development. Many issues arise due to missing dependencies or version mismatch of dependencies, which stem from environmental differences between the on-premises and cloud environments. In general, you should always use dependency managers for languages such as Maven or npm. Containers have drastically reduced dependency-based issues because all dependencies are packaged inside a container, and as such should be declared in the Dockerfile. Chef, Puppet, Ansible, and Terraform are great tools to manage and install system dependencies.

3. Configuration

   Store configuration in the environment.

   Configuration should be strictly separated from code. This allows you to easily apply configurations per environment. For example, you can have a test configuration file that stores all the connection strings and other information used in a test environment. If you want to deploy the same application to a production environment, you need only to replace the configuration. Many modern platforms support external configuration, whether it is configuration maps with Kubernetes or managed configuration services in cloud environments.

4. Backing Services

Treat backing services as attached resources.

A backing service is defined as “any service the app consumed over the network as part of its normal operation.” In the case of cloud native applications, this might be a managed caching service or a Database as a Service (DbaaS) implementation. The recommendation here is to access those services through configuration settings stored in external configuration systems, which allows loose coupling, one of the principles that is also valid for cloud native applications.

5. Build, Release, Run

   Strictly separate build and run stages.

   As you will see in Chapter 5 on DevOps, it is recommended to aim for fully automated build and release stages using CI/CD practices.

6. Processes

   Execute the app in one or more stateless processes.

   As mentioned earlier, compute in the cloud should be stateless, meaning that data should only be saved outside the processes. This enables elasticity, which is one of the promises of cloud computing.

7. Data Isolation

   Each service manages its own data.

   This is one of the key tenets of microservices architectures, which is a common pattern in cloud native applications. Each service manages its own data, which can be accessed only through APIs, meaning that other services that are part of the application are not allowed to directly access the data of another service.

8. Concurrency

   Scale out via the process model.

Improved scale and resource usage are two of the key benefits of cloud native applications, meaning that you can scale each service or function independently and horizontally; thus, you'll achieve better resource usage.

9. Disposability

   Maximize robustness with fast startup and graceful shutdown.

   Containers and functions already satisfy this factor given that both provide fast startup times. One thing that is often neglected is to design for a crash or scale in scenario, meaning that the instance count of a function or a container is decreased, which is also captured in this factor.

10. Dev/Prod Parity

    Keep development, staging, and production as similar as possible.

    Containers allow you to package all of the dependencies of your service, which limits the issues with environment inconsistencies. There are scenarios that are a bit trickier, especially when you use managed services that are not available on-premises in your Dev environment. Chapter 5 looks at methods and techniques to keep your environments as consistent as possible.

11. Logs

    Treat logs as event streams.

    Logging is one of the most important tasks in a distributed system. There are so many moving parts and without a good logging strategy, you would be "flying blind" when the application is not behaving as expected. The Twelve-Factor manifesto states that you should treat logs as streams, routed to external systems.

12. Admin Processes

    Run admin and management tasks as one-off processes.

> This basically means that you should execute administrative and management tasks as short-lived processes. Both functions and containers are great tools for that.

Throughout the book you will recognize many of these factors because they are still very relevant for cloud native applications.

Availability and Service-Level Agreements

Most of the time, cloud native applications are composite applications that use compute, such as containers and functions, but also managed cloud services such as DbaaS, caching services, and/or identity services. What is not obvious is that your compound Service-Level Agreement (SLA) will never be as high as the highest availability of an individual service. SLAs are typically measured in uptime in a year, more commonly referred to as "number of nines." Table 1-1 shows a list of common availability percentages for cloud services and their corresponding downtimes.

Table 1-1. Uptime percentages and service downtime

Availability %	Downtime per year	Downtime per month	Downtime per week
99%	3.65 days	7.20 hours	1.68 hours
99.9%	8.76 hours	43.2 minutes	10.1 minutes
99.99%	52.56 minutes	4.32 minutes	1.01 minutes
99.999%	5.26 minutes	25.9 seconds	6.05 seconds
99.9999%	31.5 seconds	2.59 seconds	0.605 seconds

Following is an example of a compound SLA:

Service 1 (99.95%) + Service 2 (99.90%): 0.9995 × 0.9990 = 0.9985005

The compound SLA is 99.85%.

Summary

Many developers struggle when starting to develop for the cloud. In a nutshell, developers are facing three major challenges: first, they need to understand distributed systems; second, they need to understand new technologies such as containers and functions; and third, they need to understand what patterns to use when building cloud native applications. Having some familiarity with the fundamentals, such as the fallacies of distributed systems, the Twelve-Factor manifesto, and compound SLAs, will make the transition easier. This chapter introduced some of the fundamental concepts of cloud native, which enables you to better understand some of the architectural considerations and patterns discussed throughout the book.

Chapter 2. Fundamentals

As discussed in Chapter 1, cloud native applications are applications that are distributed in nature and utilize cloud infrastructure. There are many technologies and tools that are being used to implement cloud native applications, but from a compute perspective, it is mainly *functions* and *containers*. From an architectural perspective, *microservices architectures* have gained a lot of popularity. More often than not, those terms are mistakenly used, and often believed to be one and the same. In reality, functions and containers are different technologies, each serving a particular purpose, whereas microservices describes an architectural style. That said, understanding how to best use functions and containers, along with *eventing* or *messaging* technologies, allows developers to design, develop, and operate a new generation of cloud native microservices-based applications in the most efficient and agile way. To make the correct architectural decisions to design those types of applications, it is important to understand the basics of the underlying terms and technologies. This chapter explains important technologies used with cloud native applications and concludes by providing an overview of the microservices architectural style.

Containers

Initially, containers were brought into the spotlight by startups and born-in-the-cloud companies, but over the past couple of years, containers have become synonymous with application modernization. Today there are very few companies that are not using containers or at least considering using

containers in the future, which means that architects and developers alike need to understand what containers offer and what they don’t offer.

When people talk about containers today, they refer to “Docker containers” most of the time, because it’s Docker that has really made containers popular. However, in the Linux operating system (OS) world, containers date back more than 10 years. The initial idea of containers was to slice up an OS so that you can securely run multiple applications without them interfering with one another. The required isolation is accomplished through namespaces and control groups, which are Linux kernel features. Namespaces allow the different components of the OS to be sliced up and thus create isolated workspaces. Control groups then allow fine-grained control of resource utilization, effectively stopping one container from consuming all system resources.

Because the interaction with kernel features was not exactly what we would call developer friendly, Linux containers (LXC) were introduced to abstract away some of the complexity of composing the various technology underpinnings of what is now commonly call a “container.” Eventually it was Docker that made containers mainstream by introducing a developer-friendly packaging of the kernel features. Docker defines containers as a “standardized unit of software.” The “unit of software”—or, more accurately, the service or application running within a container—has full, private access to their own isolated view of OS constructs. In other words, you can view containers as encapsulated, individually deployable components running as isolated instances on the same kernel with virtualization happening on the OS level.

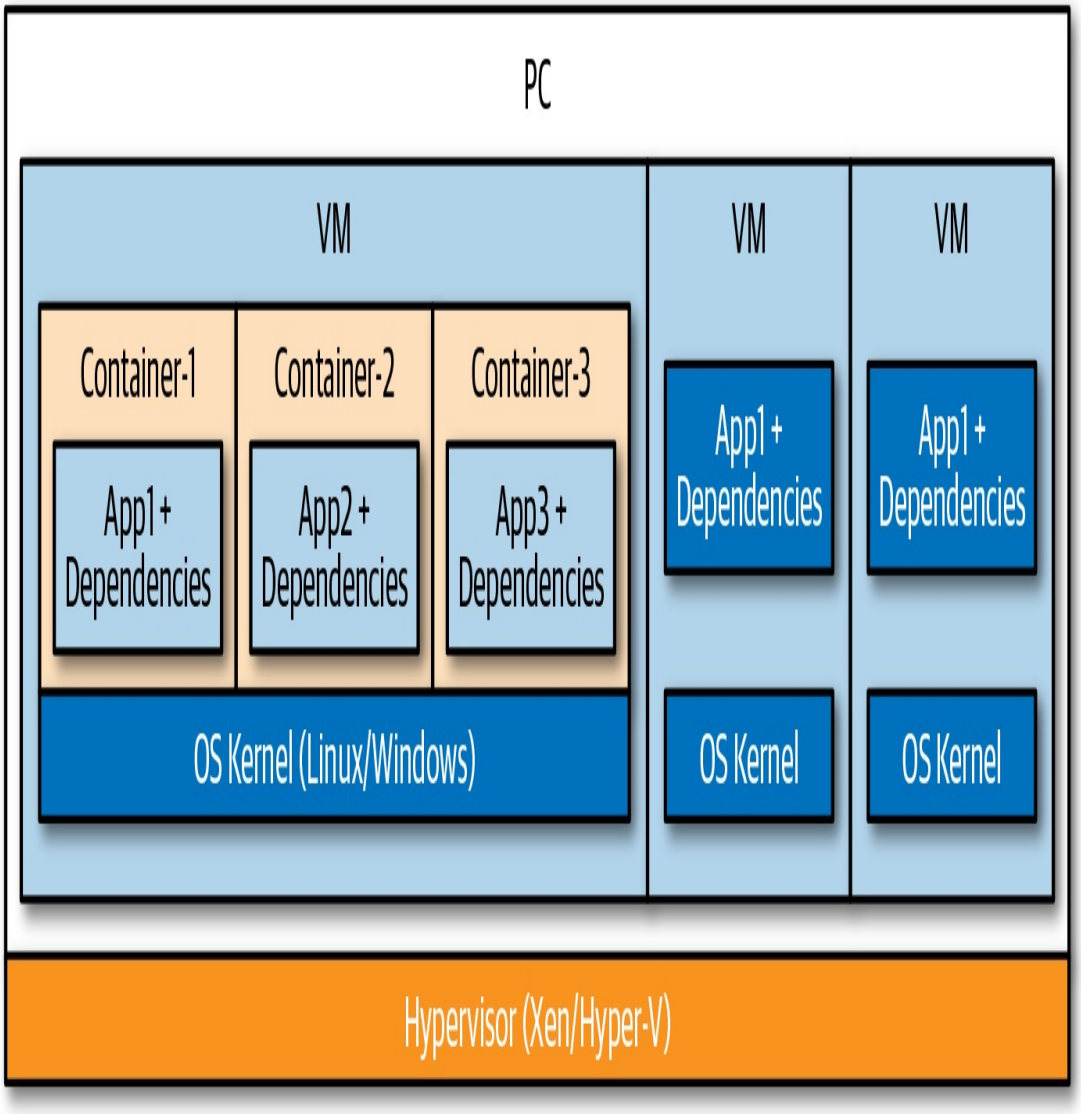

Figure 2-1. VMs and containers on a single host

In addition, containers use the copy-on-write filesystem strategy, which allows multiple containers to share the same data, and the OS provides a copy of the data to the container that needs to modify or write data. This allows containers to be very lightweight in terms of memory and disk space usage, resulting in faster startup times, which is one of the great benefits of using containers. Other benefits are *deterministic deployments*, allowing portability between environments, isolation, and higher density. For modern cloud native applications, container images have become the

unit of deployment encapsulating the application or service code, its runtime, dependencies, system libraries, and so on. Due to their fast startup times, containers are an ideal technology for scale-out scenarios, which are very common in cloud native applications. Figure 2-1 shows the difference between virtual machines (VMs) and containers on a single host.

Container Isolation Levels

Because containers are based on OS virtualization, they share the same kernel when running on the same host. Although this is sufficient enough isolation for most scenarios, it falls short of the isolation level that hardware-based virtualization options such as VMs provide. Following are some of the downsides of using VMs as the foundation of cloud native applications:

- VMs can take a considerable amount of time to start because they boot a full OS.
- The size of the VM can be an issue. A VM contains an entire OS, which can easily be several gigabytes in size. Copying this image across a network—for example, if they are kept in a central image repository—will take a lot of time.
- Scaling of VMs has its challenges. Scaling up (adding more resources) requires a new, larger VM (more CPU, memory, storage, etc.) to be provisioned and booted. Scaling out might not be fast enough to respond to demand; it takes time for new instances to start.
- VMs have more overhead and use considerably more resources such as memory, CPU, and disk. This limits the density, or number of VMs that can run on a single host machine.

The most common scenarios that demand high isolation on a hardware

virtualization level are hostile multitenant scenarios in which you typically need to protect against malicious escape and breakout attempts into other targets on the same host or on the shared infrastructure. Cloud providers have been using technologies internally that provide VM-level isolation while maintaining the expected speed and efficiency of containers. These technologies are known as *Hyper-V containers*, *sandboxed containers*, or *MicroVMs*. Here are the most popular MicroVM technologies (in nonspecific order):

Nabla containers

: These enable better isolation by taking advantage of unikernel techniques, specifically those from the Solo5 project, to limit system calls from the container to host kernel. The Nabla container runtime (runc) is an Open Container Initiative (OCI)-compliant runtime. OCI will be explained in a bit more detail later in this chapter.

Google's gVisor

: This is a container runtime and user space kernel written in Go. The new kernel is a "user space" process that addresses the container's system call needs, preventing direct interaction with the host OS. The gVisor runtime (runSC) is an OCI-compliant runtime, and it supports Kubernetes orchestration as well.

Microsoft's Hyper-V containers

: Microsoft's Hyper-V containers were introduced a couple of years ago and are based on VM Worker Process (*vmwp.exe*). Those containers provide full VM-level isolation and are OCI compliant. As for running Hyper-V containers in Kubernetes in production, you will want to wait for general availability of Kubernetes on Windows.

Kata containers

: Kata containers are a combination of Hyper.sh and Intel's clear containers and provide classic hardware-assisted virtualization. Kata

containers are compatible with the OCI specification for Docker containers and CRI for Kubernetes.

Amazon's Firecracker

Firecracker is powering Amazon's Lambda infrastructure and has been open sourced under the Apache 2.0 license. Firecracker is a user-mode VM solution that sits on top of the KVM API and is designed to run modern Linux kernels. The goal of Firecracker is to provide support for running Linux containers in a hypervisor-isolated fashion similar to other more isolated container technologies such as Kata containers. Note that, as of this writing, you are not able to use Firecracker with Kubernetes, Docker, or Kata containers.

Figure 2-2 provides an overview of the isolation levels of the these technologies.

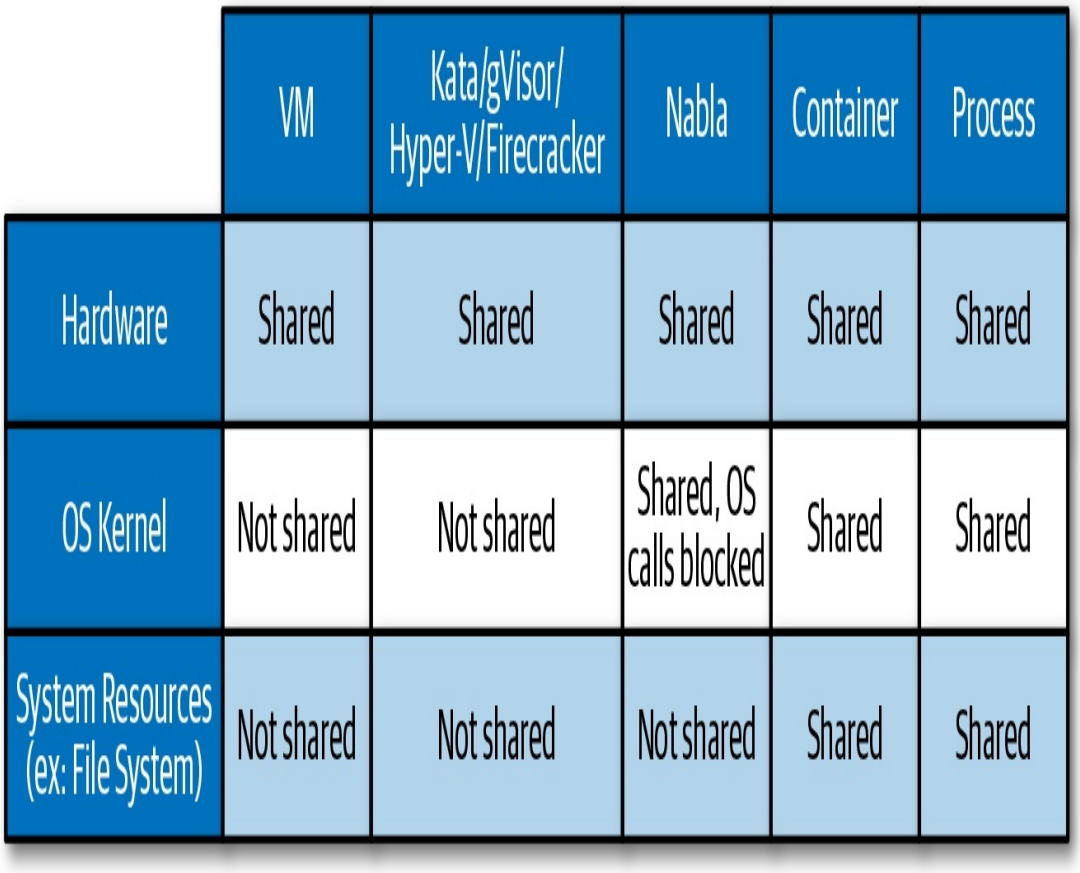

Figure 2-2. Isolation levels for VMs, containers, and processes

Container Orchestration

To manage the life cycle of containers at scale, you need to use a container orchestrator. The tasks of a container orchestrator are the following:

- The provisioning and deployment of containers onto the cluster nodes
- Resource management of containers, meaning placing containers on nodes that provide sufficient resources or moving containers to other nodes if the resource limits of a node is reached
- Health monitoring of the containers and the nodes to initiaing restarted and rescheduling in case of failures on a container or node level
- Scaling in or out containers within a cluster
- Providing mappings for containers to connect to networking
- Internal load balancing between containers

There are multiple container orchestrators available, but there is no doubt that Kubernetes is by far the most popular choice for cluster management and the scheduling of container-centric workloads in a cluster.

Kubernetes Overview

Kubernetes (often abbreviated as k8s) is an open source project for running and managing containers. Google open sourced the project in 2014, and Kubernetes is often viewed as a container platform, microservices platform, and/or a cloud portability layer. All of the major cloud vendors have a managed Kubernetes offering today.

A Kubernetes cluster runs multiple components that can be grouped in one of three categories: *master components*, *node components*, or *addons*.

Master components provide the cluster control plane. These components are responsible for making cluster-wide decisions like scheduling tasks in the cluster or responding to events, such as starting new tasks if one fails or does not meet the desired number of replicas. The master components can run on any node in the cluster, but are commonly deployed to dedicated master nodes. Managed Kubernetes offerings from cloud providers will handle the management of the control plane, including on-demand upgrades and patches.

Kubernetes master components include the following:

kube-apiserver

: Exposes the Kubernetes API and is the frontend for the Kubernetes control plane

etcd

: A key/value store used for all cluster data

kube-scheduler

: Monitors newly created *pods* (a Kubernetes-specific management wrapper around containers, which we explain in more detail later in this chapter) that are not assigned to a node and finds an available node

kube-controller-manager

: Manages a number of controllers that are responsible for responding to nodes that go down or maintaining the correct number of replicas

cloud-controller-manager

: Run controllers that interact with the underlying cloud providers

Node components run on every node in the cluster, which is also referred to as the *data plane*, and are responsible for maintaining running pods and

the environment for the node to which they are deployed.

Kubernetes node components include the following:

kubelet

: An agent that runs on each node in the cluster and is responsible for running containers in pods based on their pod specification

kube-proxy

: Maintains network rules on the nodes and performs connection forwarding

container runtime

: The software responsible for running containers (see “Kubernetes and Containers”)

Figure 2-3 shows the Kubernetes master and worker node components.

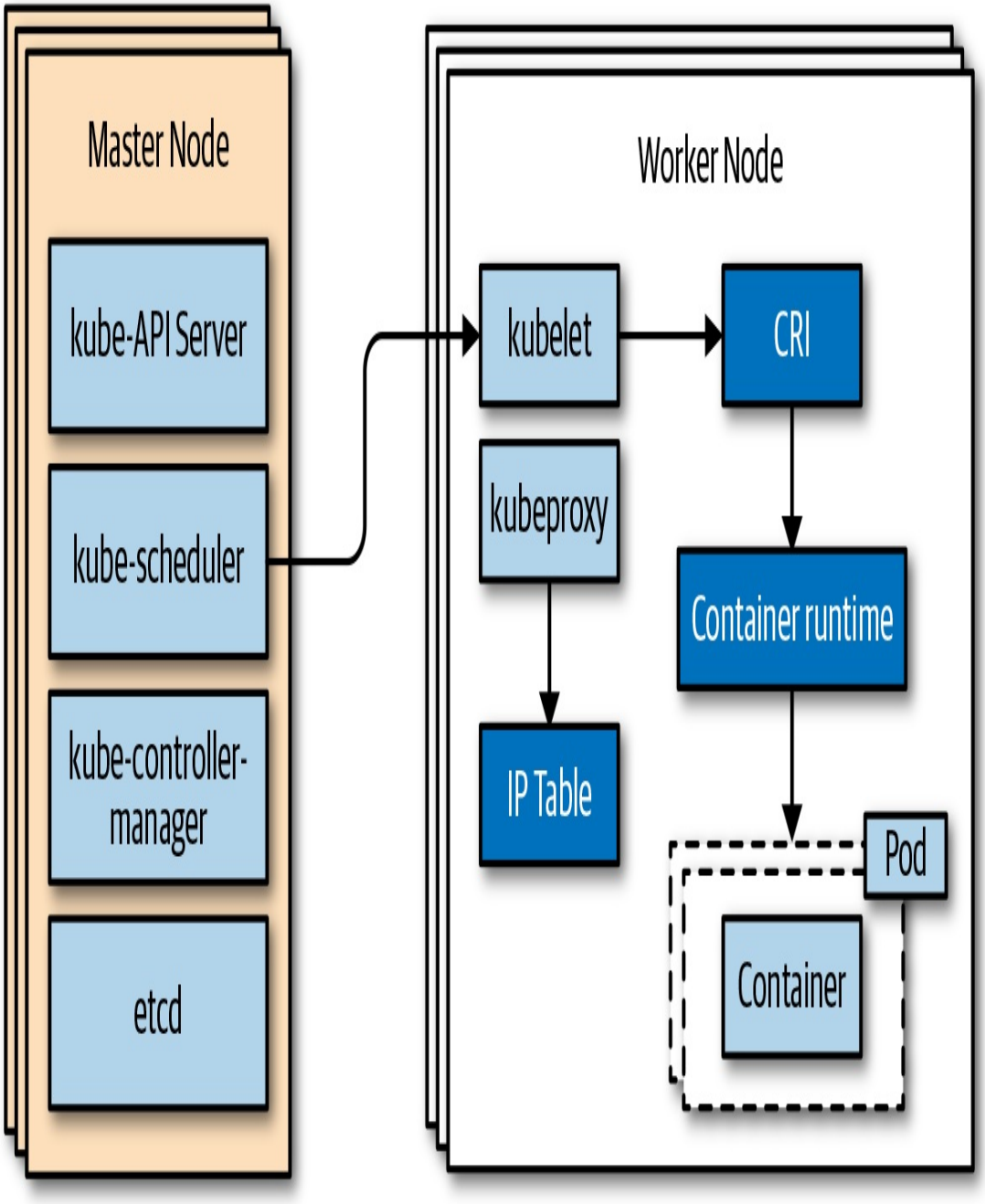

Figure 2-3. Kubernetes master and worker node components

Kubernetes is commonly deployed with addons that are managed by the master and worker node components. These addons will include services like Domain Name System (DNS) and a management user interface (UI).

A deep dive into Kubernetes is beyond the scope of this book. There are,

however, some fundamental concepts that are important for you to understand:

Pods

: A pod is basically a management wrapper around one or multiple containers, storage resources, or a unique network IP, that governs the container life cycle. Although Kubernetes supports multiple containers per pod, most of the time there is only one application container per pod. That said, the pattern of *sidecar containers*, which extends or enhances the functionality of the application container, is very popular. Service meshes like Istio rely heavily on sidecars, as you can see in Chapter 3.

Services

: A Kubernetes service provides a steady endpoint to a grouping of pods that are running on the cluster. Kubernetes uses label selectors to identify which pods are targeted by a service.

ReplicaSets

: The easiest way to think about ReplicaSets is to think about service instances. You basically define how many replicas of a pod you need, and Kubernetes makes sure that you have that number of replicas running at any given time.

Deployments

: The Kubernetes Deployment documentation states that you "describe a desired state in a Deployment object, and the Deployment controller changes the actual state to the desired state at a controlled rate." In other words, you should use Deployments for rolling out and monitoring ReplicaSets, scaling ReplicaSets, updating pods, rolling back to earlier Deployments versions, and cleaning up older ReplicaSets.

Figure 2-4 provides a logical view of the fundamental Kubernetes

concepts and how they interact with one another.

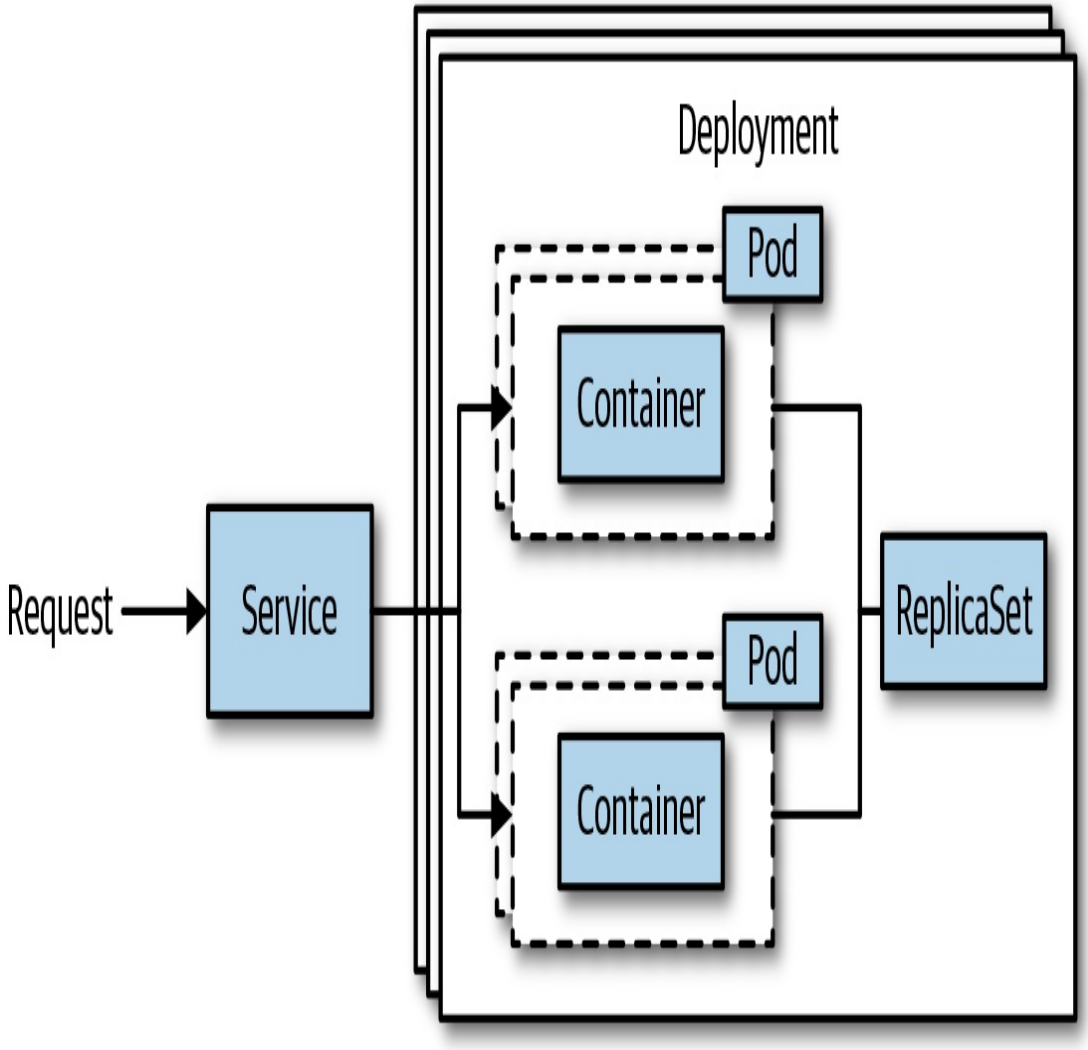

Figure 2-4. Fundamental Kubernetes concepts

Kubernetes and Containers

Kubernetes is simply the orchestration platform for containers, so it needs a container runtime to manage the container life cycle. The Docker runtime was supported from day one in Kubernetes, but it isn’t the only container runtime available on the market. As a consequence, the Kubernetes community has pushed for a generic way to integrate container

runtimes into Kubernetes. Interfaces have proven to be a good software pattern for providing contracts between two systems, so the community created the Container Runtime Interface (CRI). The CRI avoids "hardcoding" specific runtime requirements into the Kubernetes codebase, with the consequence of always needing to update the Kubernetes codebase when there are changes to a container runtime. Instead, the CRI describes the functions that need to be implemented by a container runtime to be CRI compliant. The functions that the CRI describes handle the life cycle of container pods (start, stop, pause, kill, delete), container image management (e.g., download images from a registry), and some helper functions around observability, such as log and metric collections and networking. Figure 2-5 shows high-level CRI example architectures for Docker and Kata containers.

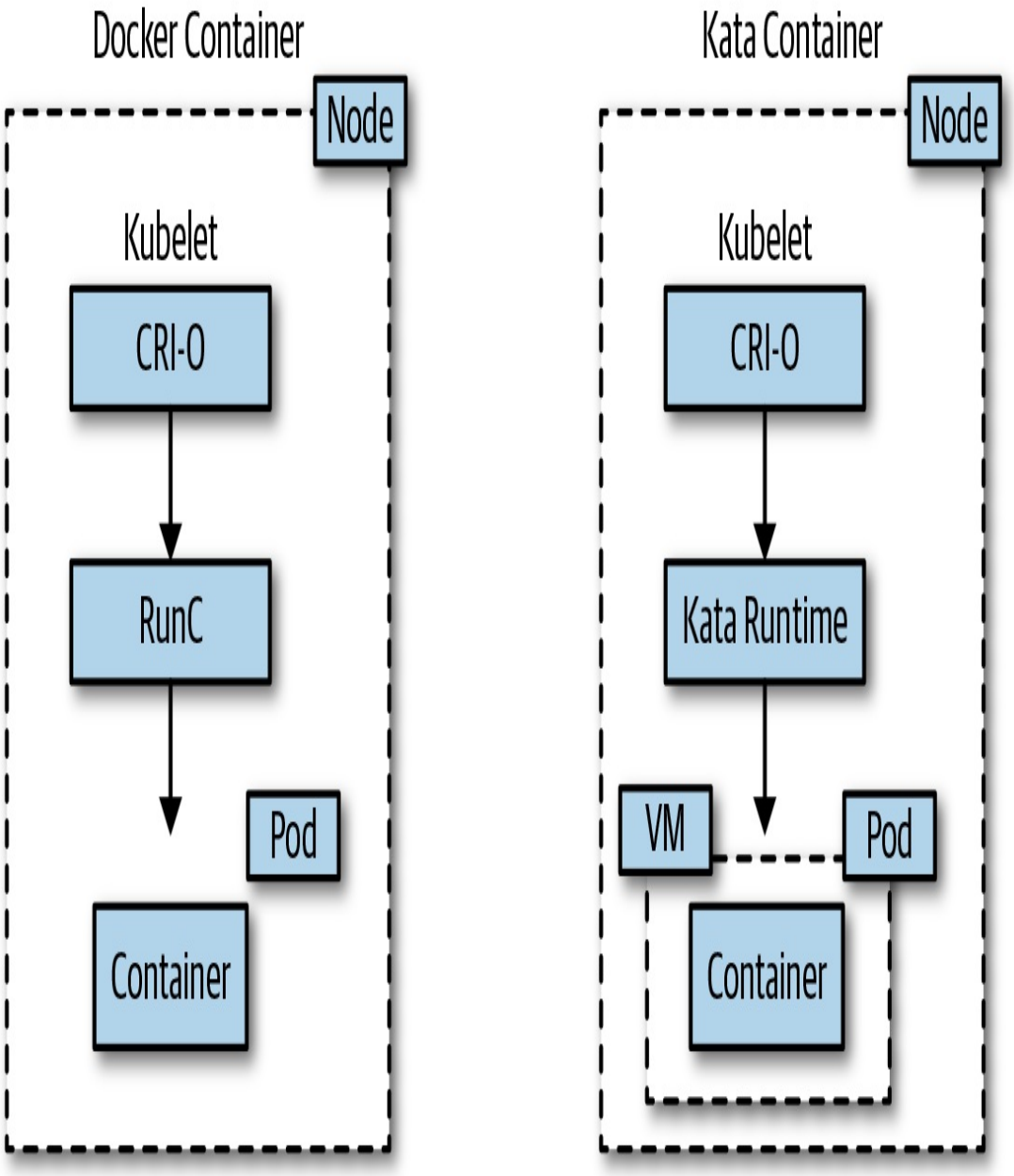

Figure 2-5. Docker versus Kata container on Kubernetes

The following list provides other container-related technologies that might be useful:

OCI

The OCI is a Linux Foundation project that aims to design open standards for container images and runtimes. Many container

technologies implement an OCI-compliant runtime and image specification.

containerd

: containerd is an industry-standard container runtime used by Docker and Kubernetes CRI, just to name the most popular ones. It is available as a daemon for Linux and Windows, which can manage the complete container life cycle of its host system, including container image management, container execution, low-level storage, and network attachments.

Moby

: Moby is a set of open source tools created by Docker to enable and accelerate software containerization. The toolkit includes container build tools, a container registry, orchestration tools, a runtime, and more, and you can use these as building blocks in conjunction with other tools and projects. Moby is using containerd as the default container runtime.

Serverless Computing

Serverless computing means that scale and the underlying infrastructure is managed by the cloud provider; that is, your application automatically drives the allocation and deallocation of resources, and you do not need to worry about managing the underlying infrastructure at all. All management and operations are abstracted away from the user and managed by cloud providers such as Microsoft Azure, Amazon Web Services (AWS), and Google Cloud Platform (GCP). From a developer perspective, serverless often adds an event-driven programming model, and from an economic perspective, you pay only per execution (CPU time consumed).

Many people think Function as a Service (FaaS) is serverless. This is

technically true, but FaaS is only one variation of serverless computing. Microsoft Azure's Container Instances (ACI) and Azure SF Mesh, as well as AWS Fargate and GCP's Serverless Containers on Cloud Functions, are good examples. ACI and AWS Fargate are serverless container offerings also known as Container as a Service (CaaS), which allow you to deploy containerized applications without needing to know about the underlying infrastructure. Other examples of serverless offerings are API management and machine learning services—basically, any service that lets you consume functionality without managing the underlying infrastructure and a pay-only-for-what-you-use model qualifies as serverless offering.

Functions

When talking about functions, people typically talk about FaaS offerings such as AWS Lambda, Azure Functions, and Google Cloud Functions, which are implemented on serverless infrastructure. The advantages of serverless computing—fast startup and execution time, plus the simplification of their applications—makes FaaS offerings very compelling to developers because it allows them to focus solely on writing code.

From a development perspective, a function is the unit of work, which means that your code has a start and a finish. Functions are usually triggered by events that are emitted by either other functions or platform services. For example, a function can be triggered by adding an entry to a database service or eventing service. There are quite a few things to consider when you want to build a large, complex application just with functions. You will need to manage more independent code, you will need to ensure state is being taken care of, and you will need to implement patterns if functions must depend on one another, just to name a few.

Containerized microservices share a lot of the same patterns, so there have been quite a few discussions around when to use FaaS or a container. Table 2-1 provides some high-level guidance between FaaS and containers, and Chapter 3 covers the trade-offs in more detail.

Table 2-1. Comparison of FaaS and containerized services

FaaS	Containerized service
Does one thing	Does more than one thing
Can’t deploy dependencies	Can deploy dependencies
Must respond to one kind of event	Can respond to more than one kind of event

There are two scenarios in which using FaaS offerings might not be ideal, although it offers the best economics. First, you want to avoid vendor lock-in. Because you need to develop your function specific to the FaaS offering and consume higher-level cloud services from a provider, your entire application becomes less portable. Second, you want to run functions on-premises or your own clusters. There are a bunch of FaaS runtimes that are available as open source runtimes and that you can run on any Kubernetes cluster. Kubeless, OpenFaaS, Serverless, and Apache OpenWhisk are among the most popular installable FaaS platforms, with Azure Functions gaining more popularity since it has been open sourced. Installable FaaS platforms are typically deployed through containers and allow the developer to simply deploy small bits of code (functions) without needing to worry about the underlying infrastructure. Many installable FaaS frameworks use Kubernetes resources for routing, autoscaling, and monitoring.

A critical aspect of any FaaS implementation, no matter whether it runs on

a cloud provider's serverless infrastructure or is installed on your own clusters, is the startup time. In general, you expect functions to execute very quickly after they have been triggered, which implies that their underlying technology needs to provide very fast boot-up times. As previously discussed, containers provide good startup times, but do not necessarily offer the best isolation.

From VMs to Cloud Native

To understand how we ended up with the next generation of cloud native applications, it is worth looking at how applications evolve from running on VMs to functions. Describing the journey should give you a good idea of how the IT industry is changing to put developer productivity into focus and how you can take advantage of all the new technologies. There are really two different paths to the cloud native world. The first one is mainly used for *brownfield scenarios*, which means that you have an existing application, and typically follows a lift-and-shift, application modernization, and eventually an application optimization process. The second one is a *greenfield scenario* in which you start your application from scratch.

Lift-and-Shift

Installing software directly on machines in the cloud is still the very first step for many customers to move to the cloud. The benefits are mainly in the capital and operational expense areas given that customers do not need to operate their own datacenters or can at least reduce operations and, therefore, the costs. From a technical perspective, lift-and-shift into Infrastructure as a Service (IaaS) gives you the most control over the entire stack. With control comes responsibility, and installing software

directly on machines often resulted in errors caused by missing dependencies, runtime versioning conflicts, resource contention, and isolation. The next logical step is to move applications into a Platform as a Service (PaaS) environment. PaaS existed long before containers became popular; for example, Azure Cloud Services dates back to 2010. In most past PaaS environments, access to the underlying VMs is restricted or in some cases prohibited so that moving to the cloud requires some rewriting of the applications. The benefit for developers is not to worry about the underlying infrastructure anymore. Operational tasks such as patching the OS were handled by the cloud providers, but some of the problems, like missing dependencies, remained. Because many PaaS services were based on VMs, scaling in burst scenarios was still a challenge due to the downsides of VMs, which we discussed previously, and for economic reasons.

Application Modernization

Besides offering super-fast startup times, containers drastically removed the issues of missing dependencies, because everything an application needed is packaged inside a container. It didn’t take long for developers to begin to love the concept of containers as a packaging format, and now pretty much every new application is using containers, and more and more monolithic legacy applications are being containerized. Many customers see the containerization of an existing application as an opportunity to also move to a more suitable architecture for cloud native environments. Microservices is the obvious choice, but as you will see later in the chapter, moving to such an architecture comes with some disadvantages. There are a few very obvious reasons, though, why you want to break up your monolith:

- Time to deployment is faster.
- Certain components need to update more frequently than others.
- Certain components need different scale requirements.
- Certain components should be developed in a different technology.
- The codebase has gotten too big and complex.

Although the methodology to break up a monolith goes beyond the scope of this book, it is worth mentioning the two major patterns to move from a monolithic application to microservices.

Strangler pattern

> With the Strangler pattern, you strangle the monolithic application. New services or existing components are implemented as microservices. A facade or gateway routes user requests to the correct application. Over time, more and more features are moved to the new architecture until the monolithic application has been entirely transformed into a microservices application.

Anticorruption Layer pattern

> This is similar to the Strangler pattern but is used when new services need to access the legacy application. The layer then translates the concepts from existing app to new, and vice versa.

We describe both patterns in more detail in Chapter 6.

With applications being packaged in container images, orchestrators began to play a more important role. Even though there were several choices in the beginning, Kubernetes has become the most popular choice today; in fact, it is considered the new cloud OS. Orchestrators, however, added another variable to the equation insomuch as development and operations teams needed to understand them. The management part of the

environment has become better, as pretty much every cloud vendor now offers “orchestrators” as a service. As with any cloud provider, “managed” Kubernetes means that the setup and runtime part of the Kubernetes service is managed. From an economical point of view, users are typically being charged for compute hours, which means that you pay as long as the nodes of the cluster are up and running even though the application might be sitting idle or utilizing low resources.

From a developer perspective, you still need to understand how Kubernetes works if you want to build microservices applications on top of it given that Kubernetes does not offer any PaaS or CaaS features out of the box.

For example, a Kubernetes service does not really represent the service code within a container, it just provides an endpoint to it, so that the code within the container can always be accessed through the same endpoint. In addition to needing to understand Kubernetes, developers are also being introduced to distributed systems patterns to handle resiliency, diagnostics, and routing, just to name a few.

Service meshes such as Istio or Linkerd are gaining popularity because they are moving some of the distributed systems complexity into the platform layer. Chapter 3 covers service meshes in great detail, but for now you can think of a service mesh as being a dedicated networking infrastructure layer that handles the service-to-service communication. Among other things, service meshes enable resiliency features such as retries and circuit breakers, distributed tracing, and routing.

The next step of application evolution is to use serverless infrastructure for containerized workloads, aka CaaS offerings such as Azure Container

Instances or AWS Fargate. Microsoft Azure has done a great job to meld the world of its managed Kubernetes Service (AKS) with its CaaS offering, ACI, by using *virtual nodes*. Virtual nodes is based on Microsoft's open source project called Virtual Kubelet, which allows any compute resource to act as a Kubernetes node and use the Kubernetes control plane. In the case of AKS virtual nodes, you are able to schedule your application on AKS and burst into ACI without needing to set up additional nodes in case your cluster cannot offer any more resources in a scale-out scenario. Figure 2-6 shows how an existing monolithic application (Legacy App) is broken down into smaller microservices (Feature 3). The legacy application and the new microservice (Feature 3) are on a service mesh on Kubernetes. In this case Feature 3 has independent scale needs and can be scaled out into a CaaS offering using Virtual Kubelet.

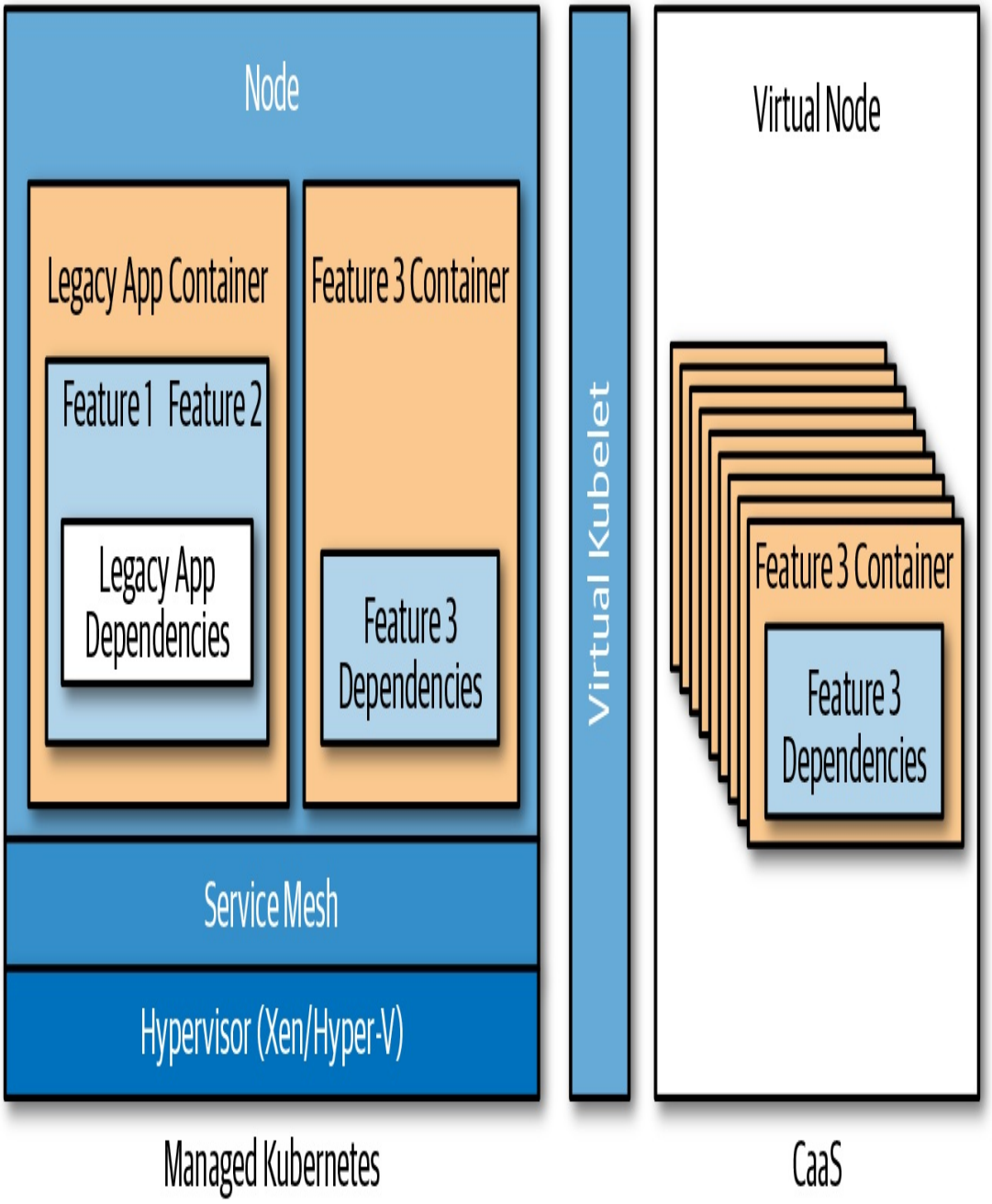

Figure 2-6. Modernized application with Feature 3 being scaled out into CaaS using Virtual Kubelet

Application Optimization

The next step is to improve the application in terms not only of further cost optimization, but also of code optimization. Functions really excel in short-lived compute scenarios, such as updating records, sending emails,

transforming messages, and so on. To take advantage of functions, you can identify short-lived compute functionality in your service codebase and implement it using functions. A good example is an order service in which the containerized microservice does all the Create, Read, Update, and Delete (CRUD) operations, and a function sends the notification of a successfully placed order. To trigger the function, eventing or messaging systems are being used. Eventually, you could decide to build the entire order service using functions, with each function executing one of the CRUD operations.

Microservices

Microservices is a term commonly used to refer to a microservices architecture style, or the individual services in a microservices architecture. A microservices architecture is a service-oriented architecture in which applications are decomposed into small, loosely coupled services by area of functionality. It’s important that services remain relatively small, are loosely coupled, and are decomposed around business capability.

Microservices architectures are often compared and contrasted with monolithic architectures. Instead of managing a single codebase, a shared datastore, and data structure, as in a monolith, in a microservices architecture an application is composed of smaller codebases, created and managed by independent teams. Each service is owned and operated by a small team, with all elements of the service contributing to a single well-defined task. Services run in separate processes and communicate through APIs that are either synchronous or asynchronous message based.

Each service can be viewed as its very own application with an

independent team, tests, builds, data, and deployments. Figure 2-7 shows the concept of a microservices architecture, using the Inventory service as an example.

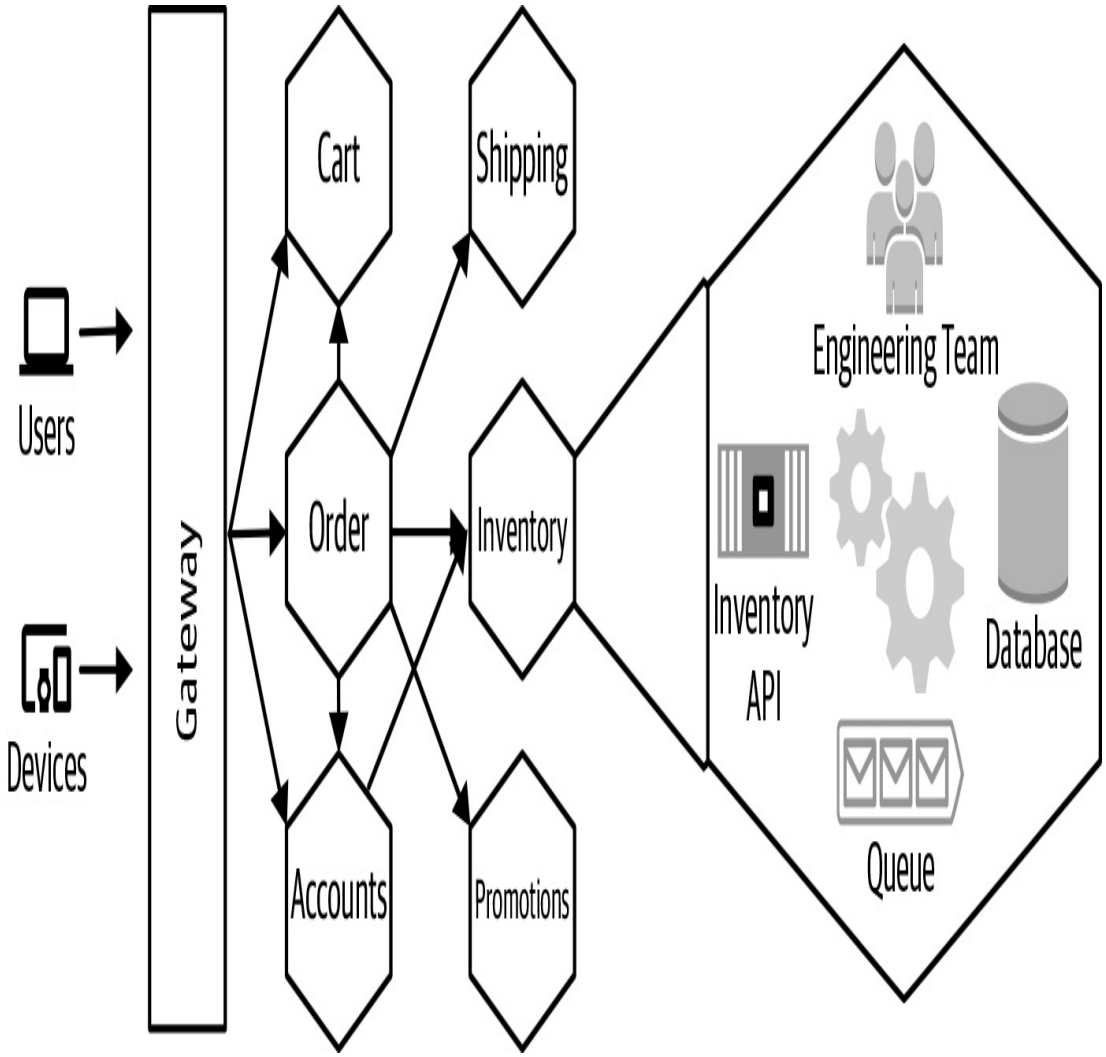

Figure 2-7. Inventory service in a microservices architecture

Benefits of a Microservices Architecture

A properly implemented microservices architecture will increase the release velocity of large applications, enabling businesses to deliver value to customers faster and more reliably.

AGILITY

Fast, reliable deployments can be challenging with large, monolithic applications. A deployment of a small change to a module in one feature area can be held up by a change to another feature. As an application grows, testing of the application will increase and it can take a considerable amount of time to deliver new value to stakeholders. A change to one feature will require the entire application to be redeployed and rolled forward or back if there is an issue with that change. By decomposing an application into smaller services, the time needed to verify and release changes can be reduced and deployed more reliably.

CONTINUOUS INNOVATION

Companies need to move increasingly faster in order to remain relevant today. This requires organizations to be agile and capable of quickly adapting to fast-changing market conditions. Companies can no longer wait years or months to deliver new value to customers: they must often deliver new value daily. A microservices architecture can make it easier to deliver value to stakeholders in a reliable way. Small independent teams are able to release features and perform A/B testing to improve conversions or user experience during even the busiest times.

EVOLUTIONARY DESIGN

With a large monolithic application, it can be very difficult to adopt new technologies or techniques because this will often require that the entire application be rewritten or care needs to be taken to ensure that some new dependency can run side-by-side with a previous one. Loose coupling and high functional cohesion is important to a system design that is able to evolve through changing technologies. By decomposing an application by features into small, loosely coupled services, it can be much easier to change individual services without affecting the entire application.

Different languages, frameworks, and libraries can be used across the different services if needed to support the business.

SMALL, FOCUSED TEAMS

Structuring engineering teams at scale and keeping them focused and productive can be challenging. Making people responsible for designing, running, and operating what they build can also be challenging if what you are building is heavily intertwined with what everyone else is building. It can sometimes take new team members days, weeks, or even months to get up to speed and begin contributing because they are burdened with understanding aspects of a system that are unrelated to their area of focus. By decomposing an application into smaller services, small agile teams are able to focus on a smaller concern and move quickly. It can be much easier for a new member joining because they need to be concerned with only a smaller service. Team members can more easily operate and take accountability for the services they build.

FAULT ISOLATION

In a monolithic application, a single library or module can cause problems for the entire application. A memory leak in one module not only can affect the stability and performance of the entire application, but can often be difficult to isolate and identify. By decomposing features of the application into independent services, teams can isolate a defect in one service to that service.

IMPROVED SCALE AND RESOURCE USAGE

Applications are generally scaled up or out. They are scaled up by increasing the size or type of machine, and scaled out by increasing the number of instances deployed and routing users across these instances.

Different features of an application will sometimes have different resource requirements; for example, memory, CPU, disk, and so on. Application features will often have different scale requirements. Some features might easily scale out with very few resources required for each instance, whereas other features might require large amounts of memory with limited ability to scale out. By decoupling these features into independent services, teams can configure the services to run in environments that best meet the services, individual resource and scale requirements.

IMPROVED OBSERVABILITY

In a monolithic application it can be difficult to measure and observe the individual components of an application without careful and detailed instrumentation throughout the application. By decomposing features of an application into separate services, teams can use tools to gain deeper insights into the behavior of the individual features and interactions with other features. System metrics such as process utilization and memory usage can now easily be tied back to the feature team because it’s running in a separate process or container.

Challenges with a Microservices Architecture

Despite all the benefits of a microservices architecture, there are trade-offs, and a microservices architecture does have its own set of challenges. Tooling and technologies have begun to address some of these challenges, but many of them still remain. A microservices architecture might not be the best choice for all applications today, but we can still apply many of the concepts and practices to other architectures. The best approach often lies somewhere in between.

COMPLEXITY

Distributed systems are inherently complex. As we decompose the application into individual services, network calls are necessary for the individual services to communicate. Networks calls add latency and experience transient failures, and the operations can run on different machines with a different clock, each having a slightly different sense of the current time. We cannot assume that the network is reliable, latency is zero, bandwidth is infinite, the network is secure, the topology will not change, there is one administrator, transport costs are zero, and that the network is homogenous. Many developers are not familiar with distributed systems and often make false assumptions when entering that world. The *Fallacies of Distributed Computing*, as discussed in Chapter 1, is a set of assertions describing those false assumptions commonly made by developers. They were first documented by L. Peter Deutsch and other Sun Microsystems engineers and are covered in numerous blog articles. Chapter 6 provides more information about best practices, tools, and techniques for dealing with the complexities of distributed systems.

DATA INTEGRITY AND CONSISTENCY

Decentralized data means that data will often exist in multiple places with relationships spanning different systems. Performing transactions across these systems can be difficult, and we need to employ a different approach to data management. One service might have a relationship to data in another service; for example, an order service might have a reference to a customer in an account service. Data might have been copied from the account service in order to satisfy some performance requirements. If the customer is removed or disabled, it can be important that the order service is updated to indicate this status. Dealing with data will require a different approach. Chapter 4 covers patterns for dealing with this.

PERFORMANCE

Networking requests and data serialization add overhead. In a microservices-based architecture the number of network requests will increase. Remember, components are libraries that are no longer making direct calls; this is happening over a network. A call to one service can result in a call to another dependent service. It might take a number of requests to multiple services in order to satisfy the original request. We can implement some patterns and best practices to mitigate potential performance overhead in a microservices architecture, which we look at in Chapter 6.

DEVELOPMENT AND TESTING

Development can be a bit more challenging because the tools and practices used today don't work with a microservices architecture. Given the velocity of change and the fact that there are many more external dependencies, it can be challenging to run a complete test suite on versions of the dependent services that will be running in production. We can implement a different approach to testing to address these challenges, and a proper Continuous Integration/Continuous Deployment (CI/CD) pipeline will be necessary. Development tooling and test strategies have evolved over the years to better accommodate a microservices architecture. Chapter 5 covers many of the tools, techniques, and best practices.

VERSIONING AND INTEGRATION

Changing an interface in a monolithic application can require some refactoring, but the changes are often built, tested, and deployed as a single cohesive unit. In a microservices architecture service, dependencies are changing and evolving independently of the consumers. Careful attention to forward and backward compatibility is necessary when

dealing with service versioning. In addition to maintaining forward and backward compatibility with service changes, it might be possible to deploy an entirely new version of the service, running it side-by-side with the previous version for some period of time. Chapter 5 explores service versioning and integration strategies.

MONITORING AND LOGGING

Many organizations struggle with monitoring and logging of monolithic applications, even when they are using a common shared logging library. Inconsistencies in naming, data types, and values make it difficult to correlate relevant log events. In a microservices architecture, when relevant events span multiple services—all potentially using different logging implementations—correlating these events can be even more challenging. Planning and early attention to the importance of logging and monitoring can help address much of this, which we examine in Chapter 5.

SERVICE DEPENDENCY MANAGEMENT

With a monolithic application, dependencies on libraries are generally compiled into a single package and tested. In a microservices architecture, service dependencies are managed differently, requiring environment-specific routing and discovery. Service discovery and routing tools and technologies have come a long way in addressing these challenges. Chapter 3 looks at these in depth.

AVAILABILITY

Although a microservices architecture can help isolate faults to individual services, if other services or the application as a whole is unable to function without that service, the application will be unavailable. As the number of services increases, the chance that one of those services

experiences a failure also increases. Services will need to implement resilient design patterns, or some functionality downgraded in the event of a service outage. Chapter 6 covers patterns and best practices for building highly available applications and provides more detail on the specific challenges.

Summary

Every application, whether cloud native or traditional, needs infrastructure on which to be hosted, technology that addresses pain points with development and deployment, and an architectural style that helps with achieving the business objectives, such as time to market. The goal of this chapter was to provide the basic knowledge for cloud native applications. By now you should understand that there are various container technologies with different isolation levels, how functions relate to containers, and that serverless infrastructure does not always need to be FaaS. Further, you should have a basic understanding of microservices architectures and of how you can migrate and modernize an existing application to be a cloud native application.

The upcoming chapters build on this knowledge and go deep into how to design, develop, and operate cloud native applications.

Chapter 3. Designing Cloud Native Applications

Application architectures are a result of unique business requirements, which makes it difficult to come up with an architectural blueprint that is generally applicable. Cloud native applications are no exception to that. A good way to approach designing cloud native applications is to consider five key areas when starting with the initial design: operational excellence, security, reliability, scalability, and cost. From an actual implementation perspective there are certain building blocks, patterns, and technologies that are proven to be very useful in solving specific problems. Besides discussing these five key areas, this chapter also covers the most common architectural building blocks.

The goal of this chapter is to equip you with the knowledge necessary to design and build cloud native architectures effectively.

Fundamentals of Cloud Native Applications

All the major cloud providers offer guidance on how to build applications targeting their respective cloud environments. Microsoft Azure has its cloud application architecture and cloud patterns guide, Amazon Web Services (AWS) has its well-architected framework, and Google offers various guides on how to build cloud native applications. Although those guides are more specific to their services offered in each environment, you can identify five generally applicable pillars that you should keep in mind regardless of the cloud provider you are choosing.

Operational Excellence ...

Chapter 4. Working with Data

Cloud computing has made a big impact on how we build and operate software today, including how we work with the data. The cost of storing data has significantly decreased, making it cheaper and more feasible for companies to keep vastly larger amounts of data. The operational overhead of database systems is considerably less with the advent of managed and serverless data storage services. This has made it easier to spread data across different data storage types, placing data into the systems better suited to manage the classification of data stored. A trend in microservices architectures encourages the decentralization of data, spreading the data for an application across multiple services, each with its own datastores. It's also common that data is replicated and partitioned in order to scale a system. Figure 4-1 shows how a typical architecture will consist of multiple data storage systems with data spread across them. It's not uncommon that data in one datastore is a copy derived from data in another store, or has some other relationship to data in another store.

Cloud native applications take advantage of managed and serverless data storage and processing services. All of the major public cloud providers offer a number of different managed services to store, process, and analyze data. In addition to cloud provider–managed database offerings, some companies provide managed databases on the cloud provider of your choice. MongoDB, for example, offers a cloud-managed database service called MongoDB Atlas that is available on Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). By using a managed database, the team can focus on building applications that use the database

instead of spending time provisioning and managing the underlying data systems.

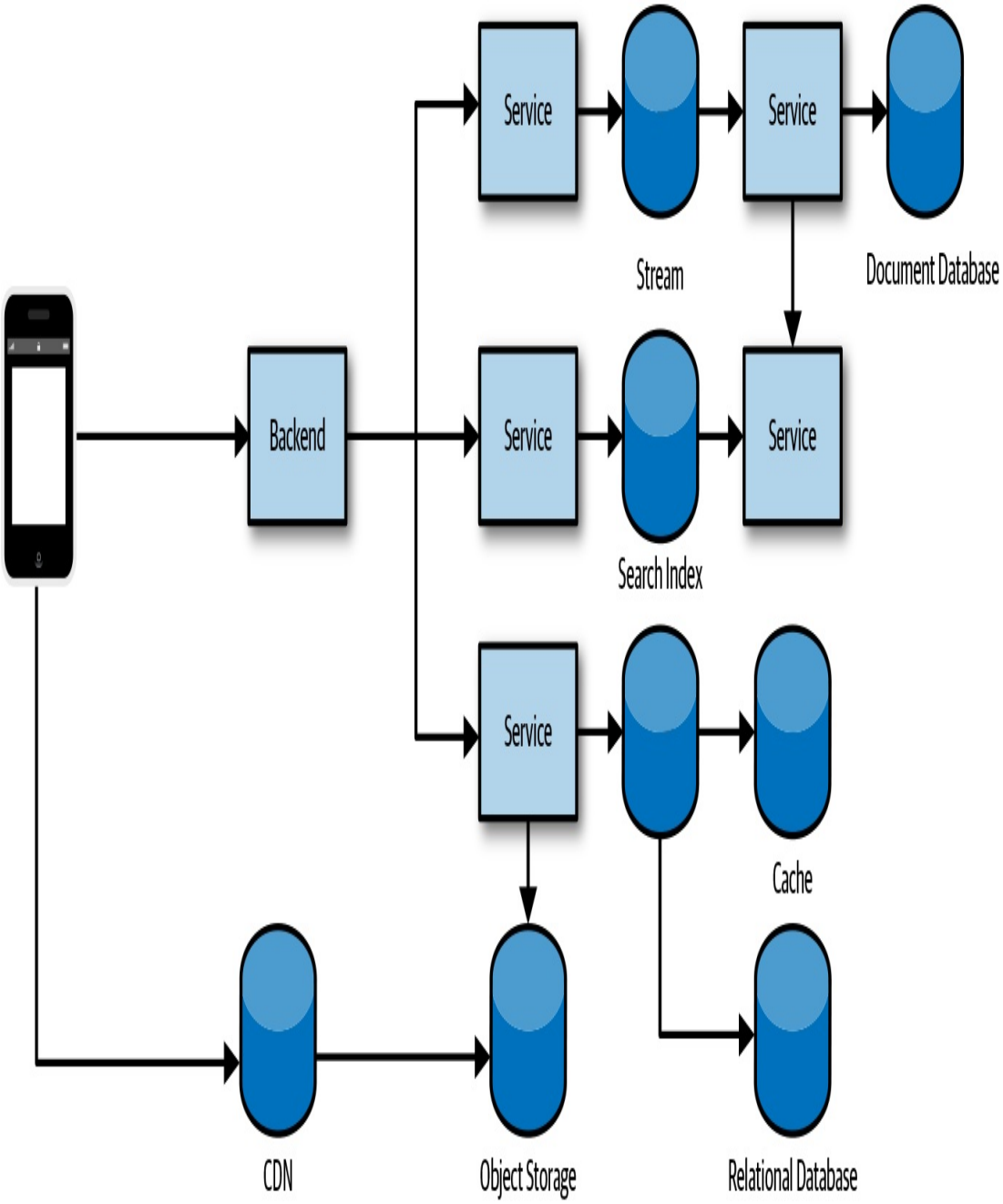

Figure 4-1. Data is often spread across multiple data systems

> **NOTE**
>
> *Serverless database* is a term that has been used to refer to a type of managed database

with usage-based billing in which customers are charged based on the amount of data stored and processed. This means that if a database is not being accessed, the user is billed only for the amount of data stored. When there is an operation on the database, either the user is charged for the specific operation or the database is scaled from zero and back during the processing of the operation.

Cloud native applications take full advantage of the cloud, including data systems used. The following is a list of cloud native application characteristics for data:

- Prefer managed data storage and analytics services.
- Use polyglot persistence, data partitioning, and caching.
- Embrace eventual consistency and use strong consistency when necessary.
- Prefer cloud native databases that scale out, tolerate faults, and are optimized for cloud storage.
- Deal with data distributed across multiple datastores.

Cloud native applications often need to deal with silos of data, which require a different approach to working with data. There are a number of benefits to polyglot persistence, decentralized data, and data partitioning, but there are also trade-offs and considerations.

Data Storage Systems

There are a growing number of options for storing and processing data. It can be difficult to determine which products to use when building an application. Teams will sometimes engage in a number of iterations evaluating languages, frameworks, and the data storage systems that will be used in the application. Many are still not convinced they made the

correct decision, and it’s common for those storage systems to be replaced or new ones added as the application evolves anyway.

It can be helpful to understand the various types of datastores and the workloads they are optimized for when deciding which products to use. Many products are, however, multimodel and are designed to support multiple data models, falling into multiple data storage classifications. Applications will often take advantage of multiple data storage systems, storing files in an object store, writing data to a relational database, and caching with an in-memory key/value store.

Objects, Files, and Disks

Every public cloud provider offers an inexpensive *object storage service*. Object storage services manage data as objects. Objects are usually stored with metadata for the object and a key that’s used as a reference for the object. File storage services generally provide shared access to files through a traditional file sharing model with a hierarchical directory structure. Disks or block storage provides storage of disk volumes used by computing instances. Determining where to store files such as images, documents, content, and genomics data files will largely depend on the systems that access them. Each of the following storage types is better suited for different types of files:

> **NOTE**
>
> You should prefer object storage for storing file data. Object storage is relatively inexpensive, extremely durable, and highly available. All of the major cloud providers offer different storage tiers enabling cost saving based on data access requirements.

Object/blob storage

- Use it with files when the applications accessing the data support the cloud provider API.
- It is inexpensive and can store large amounts of data.
- Applications need to implement a cloud provider API. If application portability is a requirement, see Chapter 7.

File storage

- Use it with applications designed to support Network Attached Storage (NAS).
- Use it when using a library or service that requires shared access to files.
- It is more expensive than object storage.

Disk (block) storage

- Use it for applications that assume persistent local storage disks, like MongoDB or a MySQL database.

In addition to the various cloud provider–managed storage options for files and objects, you can provision a distributed filesystem. The Hadoop Distributed File System (HDFS) is popular for big data analytics. The distributed filesystem can use the cloud provider disk or block storage services. Many of the cloud providers have managed services for popular distributed filesystems that include the analytics tools used. You should consider these filesystems when using the analytics tools that work with them.

Databases

Databases are generally used for storing more structured data with well-

defined formats. A number of databases have been released over the past few years, and the number of databases available for us to choose from continues to grow every year. Many of these databases have been designed for specific types of data models and workloads. Some of them support multiple models and are often labeled as *multimodel databases*. It helps to organize databases into a group or classification when considering which database to use where in an application.

KEY/VALUE

Often, application data needs to be retrieved using only the primary key, or maybe even part of the key. A key/value store can be viewed as simply a very large hash table that stores some value under a unique key. The value can be retrieved very efficiently using the key or, in some cases, part of the key. Because the value is opaque to the database, a consumer would need to scan record-by-record in order to find an item based on the value. The keys in a key/value database can comprise multiple elements and even can be ordered for efficient lookup. Some of the key/value databases allow for the lookup using the key prefix, making it possible to use compound keys. If the data can be queried based on some simple nesting of keys, this might be a suitable option. If we're storing orders for customer xyz in a key/value store, we might store them using the customer ID as a key prefix followed by the order number, "xyz-1001." A specific order can be retrieved using the entire key, and orders for customer xyz could be retrieved using the "xyz" prefix.

NOTE

Key/value databases are generally inexpensive and very scalable datastores. Key/value data storage services are capable of partitioning and even repartitioning data based on the key. Selecting a key is important when using these datastores because it will have a significant impact on the scale and the performance of data storage reads and writes.

DOCUMENT

A document database is similar to a key/value database in that it stores a document (value) by a primary key. Unlike a key/value database, which can store just about any value, the documents in a document database need to conform to some defined structure. This enables features like the maintenance of secondary indexes and the ability to query data based on the document. The values commonly stored in a document database are a composition of hashmaps (JSON objects) and lists (JSON arrays). JSON is a popular format used in document databases, although many database engines use a more efficient internal storage format like MongoDB's BSON.

> **TIP**
>
> You will need to think differently about how you organize data in a document-oriented database when coming from relational databases. It takes time for many to make the transition to this different approach to data modeling.

You can use these databases for much of what was traditionally stored in a relational database like PostgreSQL. They have been growing in popularity and unlike wwith relational databases, the documents map nicely to objects in programming languages and don't require object relational mapping (ORM) tools. These databases generally don't enforce a schema, which has some advantages with regard to Continuous Delivery (CD) of software changes requiring data schema changes.

> **NOTE**

Databases that do not enforce a schema are often referred to “schema on read” because although the database does not enforce the schema, an inherent schema exists in the applications consuming the data and will need to know how to work with the data returned.

RELATIONAL

Relational databases organize data into two-dimensional structures called tables, consisting of columns and rows. Data in one table can have a relationship to data in another table, which the database system can enforce. Relational databases generally enforce a strict schema, also referred to *schema on write*, in which a consumer writing data to a database must conform to a schema defined in the database.

Relational databases have been around for a long time and a lot of developers have experience working with them. The most popular and commonly used databases, as of today, are still relational databases. These databases are very mature, they’re good with data that contains a large number of relationships, and there’s a large ecosystem of tools and applications that know how to work with them. *Many-to-many relationships* can be difficult to work with in document databases, but in relational database they are very simple. If the application data has a lot of relationships, especially those that require transactions, these databases might be a good fit.

GRAPH

A graph database stores two types of information: *edges* and *nodes*. Edges define the relationships between nodes, and you can think of a node as the entity. Both nodes and edges can have properties providing information about that specific edge or node. An edge will often define the direction or

nature of a relationship. Graph databases work well at analyzing the relationships between entities. Graph data can be stored in any of the other databases, but when graph traversal becomes increasingly complex, it can be challenging to meet the performance and scale requirements of graph data in the other storage types.

COLUMN FAMILY

A column-family database organizes data into rows and columns, and can initially appear very similar to a relational database. You can think of a column-family database as holding tabular data with rows and columns, but the columns are divided into groups known as column families. Each column family holds a set of columns that are logically related together and are typically retrieved or manipulated as a unit. Other data that is accessed separately can be stored in separate column families. Within a column family, new columns can be added dynamically, and rows can be sparse (that is, a row doesn’t need to have a value for every column).

TIME-SERIES

Time-series data is a database that’s optimized for time, storing values based on time. These databases generally need to support a very high number of writes. They are commonly used to collect large amounts of data in real time from a large number of sources. Updates to the data are rare and deletes are often completed in bulk. The records written to a time-series database are usually very small, but there are often a large number of records. Time-series databases are good for storing telemetry data. Popular uses include Internet of Things (IoT) sensors or application/system counters. Time-series databases will often include features for data retention, down-sampling, and storing data in different mediums depending on configuration data usage patterns.

SEARCH

Search engine databases are often used to search for information held in other datastores and services. A search engine database can index large volumes of data with near-real-time access to the indexes. In addition to searching across unstructured data like that in a web page, many applications use them to provide structured and ad hoc search features on top of data in another database. Some databases have full-text indexing features, but search databases are also capable of reducing words to their root forms through stemming and normalization.

Streams and Queues

Streams and queues are data storage systems that store events and messages. Although they are sometimes used for the same purpose, they are very different types of systems. In an event stream, data is stored as an immutable stream of events. A consumer is able to read events in the stream at a specific location but is unable to modify the events or the stream. You cannot remove or delete individual events from the stream. Messaging queues or topics will store messages that can be changed (mutated), and it's possible to remove an individual message from a queue. Streams are great at recording a series of events, and streaming systems are generally able to store and process very large amounts of data. Queues or topics are great for messaging between different services, and these systems are generally designed for the short-term storage of messages that can be changed and randomly deleted. This chapter focuses more on streams because they are more commonly used with data systems, and queues more commonly used for service communications. For more information on queues, see Chapter 3.

NOTE

A topic is a concept used in a publish-subscribe messaging model. The only difference between a topic and a queue is that a message on a queue goes to one subscriber, whereas a message to a topic will go to multiple subscribers. You can think of a queue as a topic with one, and only one, subscriber.

Blockchain

Records on a blockchain are stored in a way that they are immutable. Records are grouped in a *block*, each of which contains some number of records in the database. Every time new records are created, they are grouped into a single block and added to the chain. Blocks are chained together using hashing to ensure that they are not tampered with. The slightest change to the data in a block will change the hash. The hash from each block is stored at the beginning of the next block, ensuring that nobody can change or remove a block from the chain. Although a blockchain could be used like any other centralized database, it's commonly decentralized, removing power from a central organization.

Selecting a Datastore

When selecting a datastore, you need to consider a number of requirements. Selecting data storage technologies and services can be quite challenging, especially given the cool new databases constantly becoming available and changes in how we build software. Start with the architecturally significant requirements—also known as *nonfunctional* requirements—for a system and then move to the functional requirements.

Selecting the appropriate datastore for your requirements can be an important design decision. There are literally hundreds of implementations to choose from among SQL and NoSQL databases. Datastores are often categorized by how they structure data and the types of operations they

support. A good place to begin is by considering which storage model is best suited for the requirements. Then, consider a particular datastore within that category, based on factors such as feature set, cost, and ease of management.

Gather as much of the following information as you can about your data requirements.

FUNCTIONAL REQUIREMENTS

Data format

: What type of data do you need to store?

Read and write

: How will the data need to be consumed and written?

Data size

: How large are the items that will be placed in the datastore?

Scale and structure

: How much storage capacity do you need, and do you anticipate needing to partition your data?

Data relationships

: Will your data need to support complex relationships?

Consistency model

: Will you require strong consistency or is eventual consistency acceptable?

Schema flexibility

: What kind of schemas will you apply to your data? Is a fixed or strongly enforced schema important?

Concurrency

Will the application benefit from multiversion concurrency control? Do you require pessimistic and/or optimistic concurrency control?

Data movement

Will your application need to move data to other stores or data warehouses?

Data life cycle

Is the data write-once, read-many? Can it be archived over time or can the fidelity of the data be reduced through down-sampling?

Change streams

Do you need to support change data capture (CDC) and fire events when data changes?

Other supported features

Do you need any other specific features, full-text search, indexing, and so on?

NONFUNCTIONAL REQUIREMENTS

Team experience

Probably one of the biggest reasons teams select a specific database solution is because of experience.

Support

Sometimes the database system that’s the best technical fit for an application is not the best fit for a project because of the support options available. Consider whether or not available support options meet the organizations needs.

Performance and scalability

What are your performance requirements? Is the workload heavy on ingestion? Query and analytics?

Reliability

What are the availability requirements? What backup and restore features are necessary?

Replication

Will data need to be replicated across multiple regions or zones?

Limits

Are there any hard limits on size and scale?

Portability

Do you need to deploy on-premises or to multiple cloud providers?

MANAGEMENT AND COST

Managed service

When possible, use a managed data service. There are, however, situations for which a feature is not available and needed.

Region or cloud provider availability

Is there a managed data storage solution available?

Licensing

Are there any restrictions on licensing types in the organization? Do you have a preference of a proprietary versus open source software (OSS) license?

Overall cost

What is the overall cost of using the service within your solution? A good reason to prefer managed services is for the reduced operational cost.

Selecting a database can be a bit daunting when you're looking across the vast number of databases available today and the new ones constantly introduced in the market. A site that tracks database popularity, db-engines (*https://db-engines.com*), lists 329 different databases as of this writing. In

many cases the skillset of the team is a major driving factor when selecting a database. Managing data systems can add significant operational overhead and burden to the team and managed data systems are often preferred for cloud-native applications, so the availability of managed data systems will quite often narrow down the options. Deploying a simple database can be easy, but consider that the patching, upgrades, performance tuning, backups, and highly available database configurations increase operations burden. Yet there are situations in which managing a database is necessary, and you might prefer some of the new databases built for the cloud, like CockroachDB or YugaByte. Also consider available tooling: it might make sense to deploy and manage a certain database if this avoids the need to build software to consume the data, like a dashboard or reporting systems.

Data in Multiple Datastores

Whether you're working with data across partitions, databases, or services, data in multiple datastores can introduce some data management challenges. Traditional transaction management might not be possible and distributed transactions will adversely affect the performance and scale of a system. The following are some of the challenges of distributing data:

- Data consistency across the datastores
- Analysis of data in multiple datastores
- Backup and restore of the datastores

The consistency and integrity of the data can be challenging when spread across multiple datastores. How do you ensure a related record in one system is updated to reflect a change in another system? How do you manage copies of data, whether they are cached in memory, a materialized

view, or stored in the systems of another service team? How do you effectively analyze data that's stored across multiple silos? Much of this is addressed through data movement, and a growing number of technologies and services are showing up in the market to handle this.

Change Data Capture

Many of the database options available today offer a stream of data change events (change log) and expose this through an easy-to-consume API. This can make it possible to perform some actions on the events, like triggering a function when a document changes or updating a materialized view. For example, successfully adding a document that contains an order could trigger an event to update reporting totals and notify an accounting service that an order for the customer has been created. Given a move to polyglot persistence and decentralized datastores, these event streams are incredibly helpful in maintaining consistency across these silos of data. Some common use cases for CDC include:

Notifications

: In a microservices architecture, it's not uncommon that another service will want to be notified of changes to data in a service. For this, you can use a webhook or subscription to publish events for other services.

Materialized views

: Materialized views make for efficient and simplified queries on a system. The change events can be used to update these views.

Cache invalidation

: Caches are great for improving the scale and performance of a system, but invalidating the cache when the backing data has changed is a challenge. Instead of using a time-to-live (TTL), you can use change events to either remove the cached item or update it.

Auditing

Many systems need to maintain a record of changes to data. You can use this log of changes to track what was changed and when. The user that made the change is often needed, so it might be necessary to ensure that this information is also captured.

Search

Many databases are not very good at handling search, and the search datastores do not provide all of the features needed in other databases. You can use change streams to maintain a search index.

Analytics

The data analytics requirements of an organization often require a view across many different databases. Moving the data to a central data lake, warehouse, or database can enable richer reporting and analytics requirements.

Change analytics

Near-real-time analysis of data changes can be separated from the data access concerns and performed on the data changes.

Archive

In some applications, it is necessary to maintain an archive of state. This archive is rarely accessed, and it's often better to store this in a less expensive storage system.

Legacy systems

Replacing a legacy system will sometimes require data to be maintained in multiple locations. These change streams can be used to update data in a legacy system.

In Figure 4-2, we see an app writing to a database that logs a change. That change is then written to a stream of change logs and processed by multiple consumers. Many database systems maintain an internal log of changes that can be subscribed to with checkpoints to resume at a specific

location. MongoDB, for example, allows you to subscribe to events on a deployment, data, or collection, and provide a token to resume at a specific location. Many of the cloud provider databases handle the watch process and will invoke a serverless function for every change.

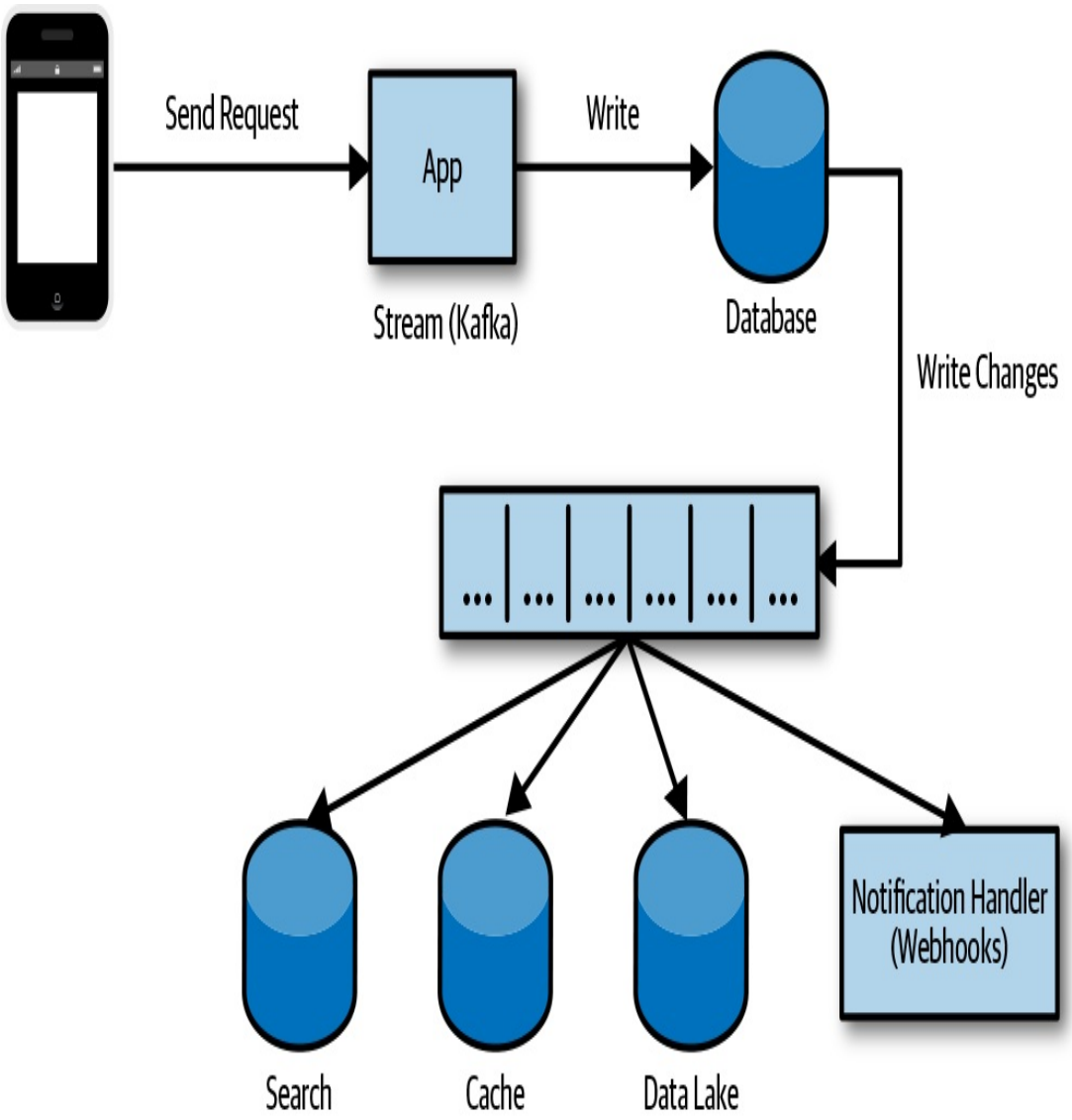

Figure 4-2. CDC used to synchronize data changes

The application could have written the change to the stream and the database, but this presents some problems if one of the two operations fails and it potentially creates a race condition. For example, if the application were updating some data in the database, like an account

shipping preference, and then failed to write to an event stream, the data in the database would have changed, but the other systems would not have been notified or updated, like a shipping service. The other concern is that if two processes made a change to the same record at close to the same time, the order to events can be a problem. Depending on the change and how it's processed, this might not be an issue, but it's something to consider. The concern is that we either record the event that something changed when it didn't, or change something and don't record the event.

By using the databases change stream, we can write the change or mutation of the document and the log of that change as a transaction. Even though data systems consuming the event stream are eventually consistent after some period of time, it's important that they become consistent. Figure 4-3 shows a document that has been updated and the change recorded as part of a transaction. This ensures that the change event and the actual change itself are consistent, so now we just need to consume and process that event into other systems.

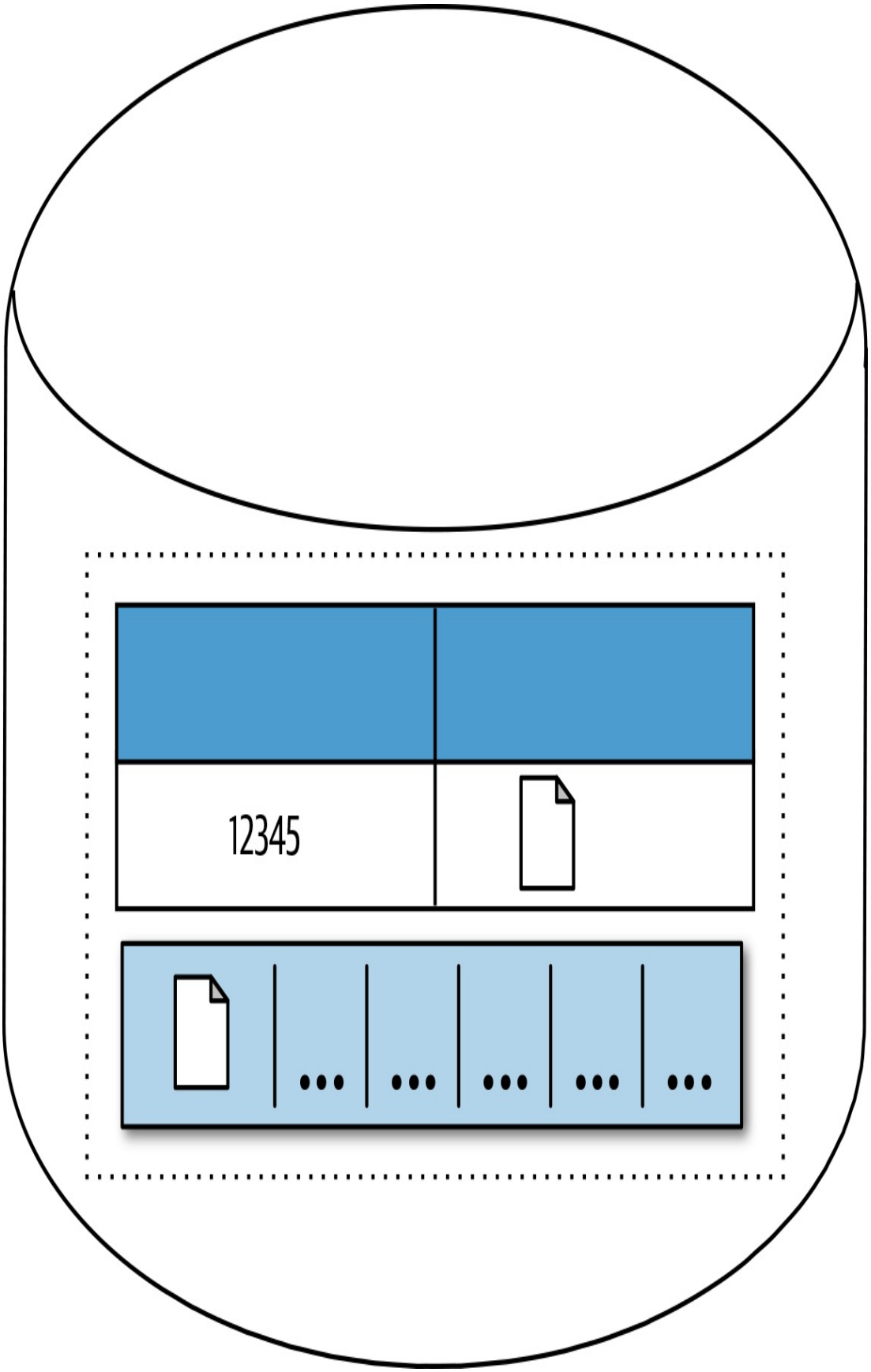
12345

Figure 4-3. Changes to a record and operation log in a transaction scope

Many of the managed data services make this really easy to implement and can be quickly configured to invoke a serverless function when a change happens in the datastore. You can configure MongoDB Atlas to invoke a function in the MongoDB Stitch service. A change in Amazon DynamoDB or Amazon Simple Storage Service (Amazon S3) can trigger a lambda function. Microsoft Azure Functions can be invoked when a change happens in Azure Cosmos DB or Azure Blob Storage. A change in Google Cloud Firestore or object storage service can trigger a Cloud Function. Implementation with popular managed data storage services can be fairly straightforward. This is becoming a popular and necessary feature with most datastores.

Write Changes as an Event to a Change Log

As we just saw an application failure during an operation that affects multiple datastores can result in data consistency issues. Another approach that you can use when an operation spans multiple databases is to write the set of changes to a change log and then apply those changes. A group of changes can be written to a stream maintaining order, and if a failure occurs while the changes are being applied, it can be easy to retry or resume the operation, as shown in Figure 4-4.

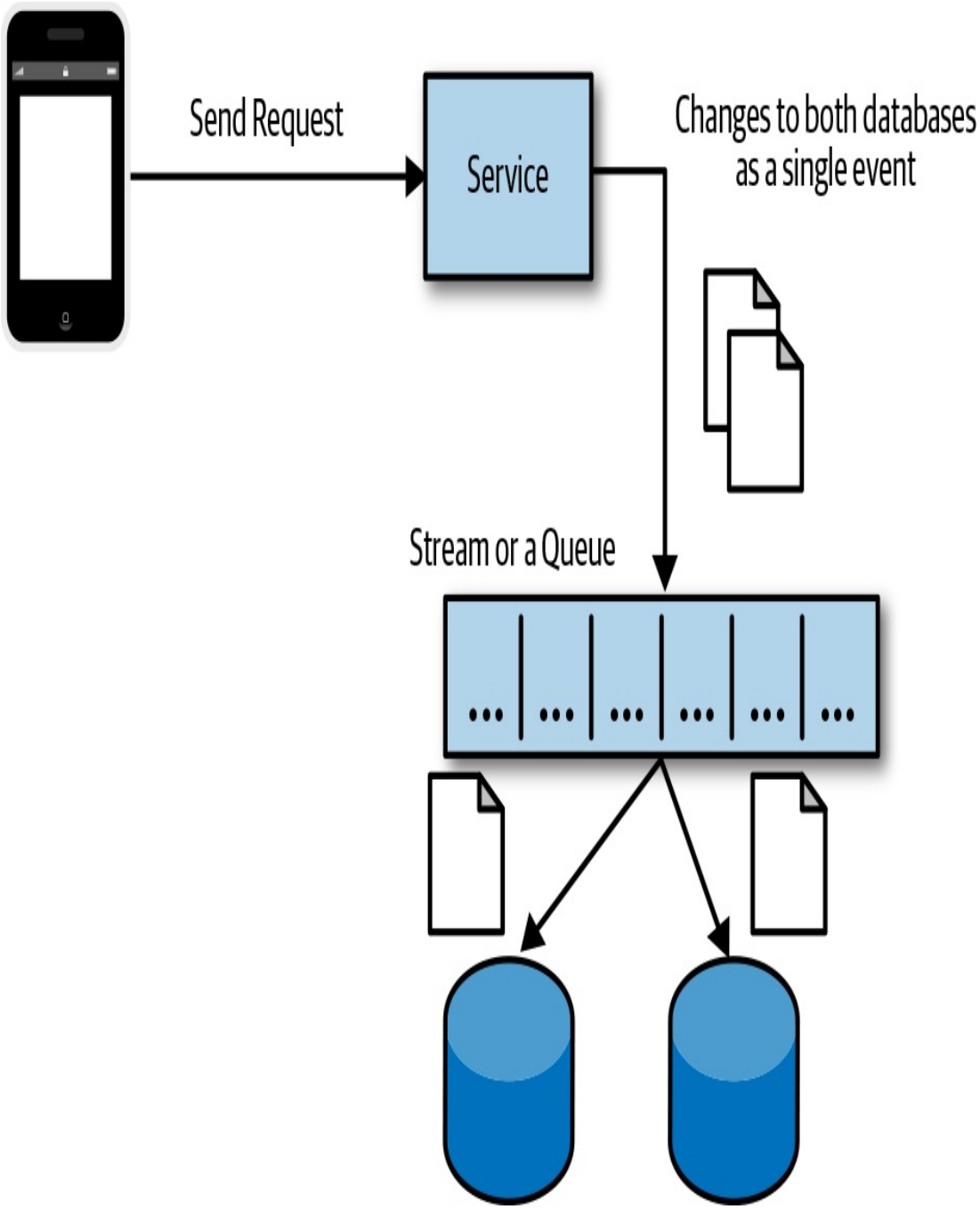

Figure 4-4. Saving a set of changes before writing each change

Transaction Supervisor

You can use a supervisor service to ensure that a transaction is successfully completed or is compensated. This can be especially useful when you're performing transactions involving external services—for

example, writing an order to the system and processing a credit card, in which credit card processing can fail, or saving the results of the processing. As Figure 4-5 illustrates, a checkout service receives an order, processes a credit card payment, and then fails to save the order to the order database. Most customers would be upset to know that their credit card was processed but there was no record of their order. This is a fairly common implementation.

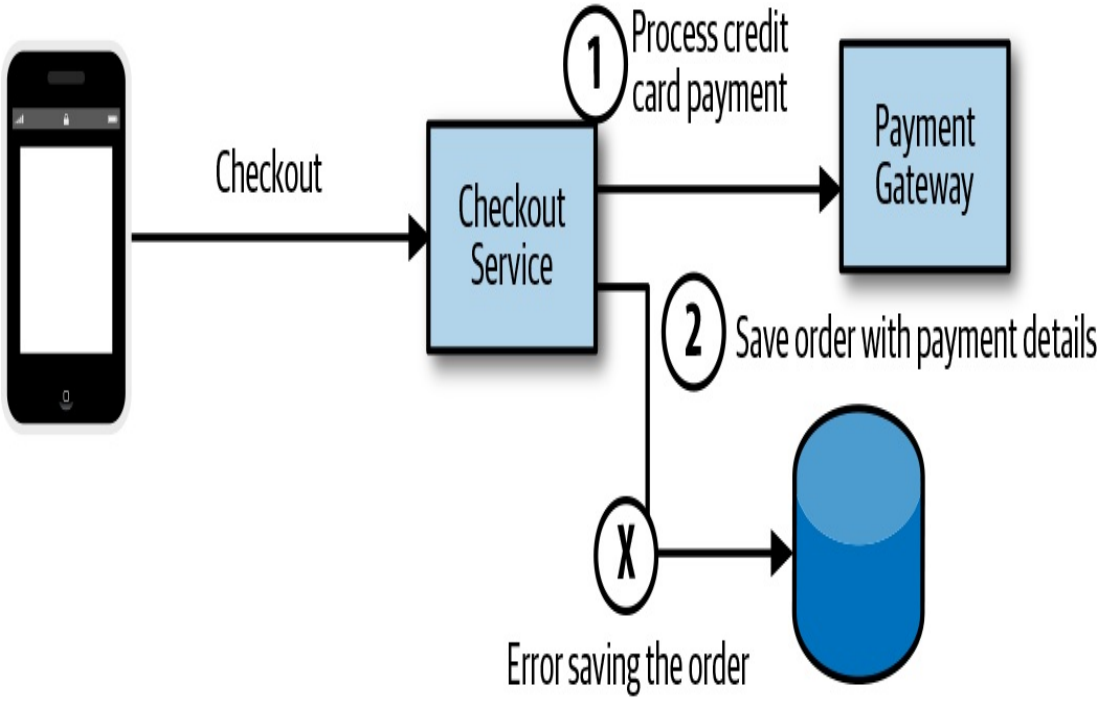

Figure 4-5. Failing to save order details after processing an order

Another approach might be to save the order or cart with a status of processing, then make the call to the payment gateway to process the credit card payment, and finally, update the status of the order. Figure 4-6 demonstrates how if we fail to update the order status, at least we have the record of an order submitted and the intention to process it. If the payment gateway service offered a notification service like a webhook callback, we could configure that to ensure that the status was accurate.

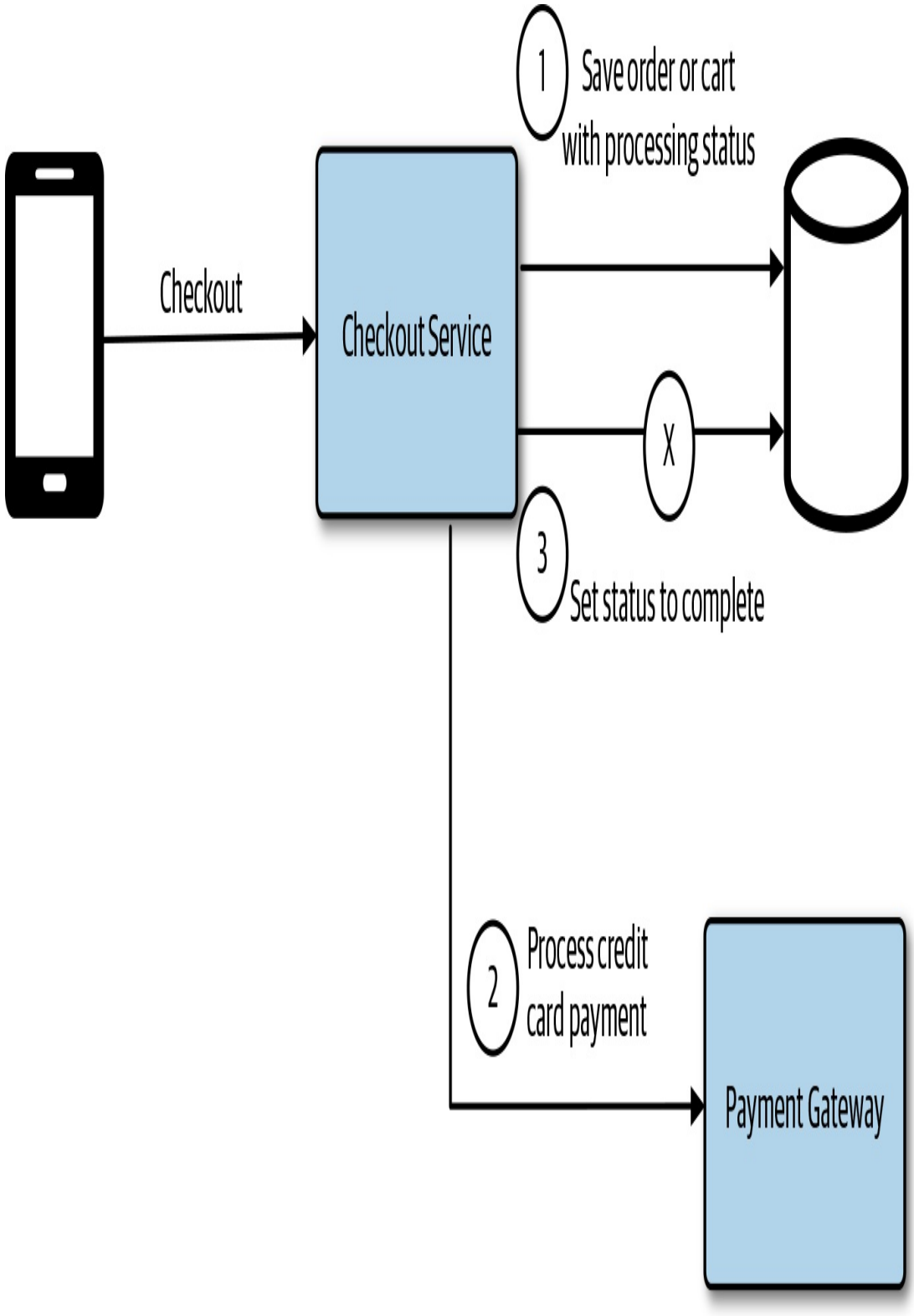

Figure 4-6. Failing to update order status

In Figure 4-7, a supervisor is added to monitor the order database for

processing transactions that have not completed and reconciles the state. The supervisor could be a simple function that's triggered at a specific interval.

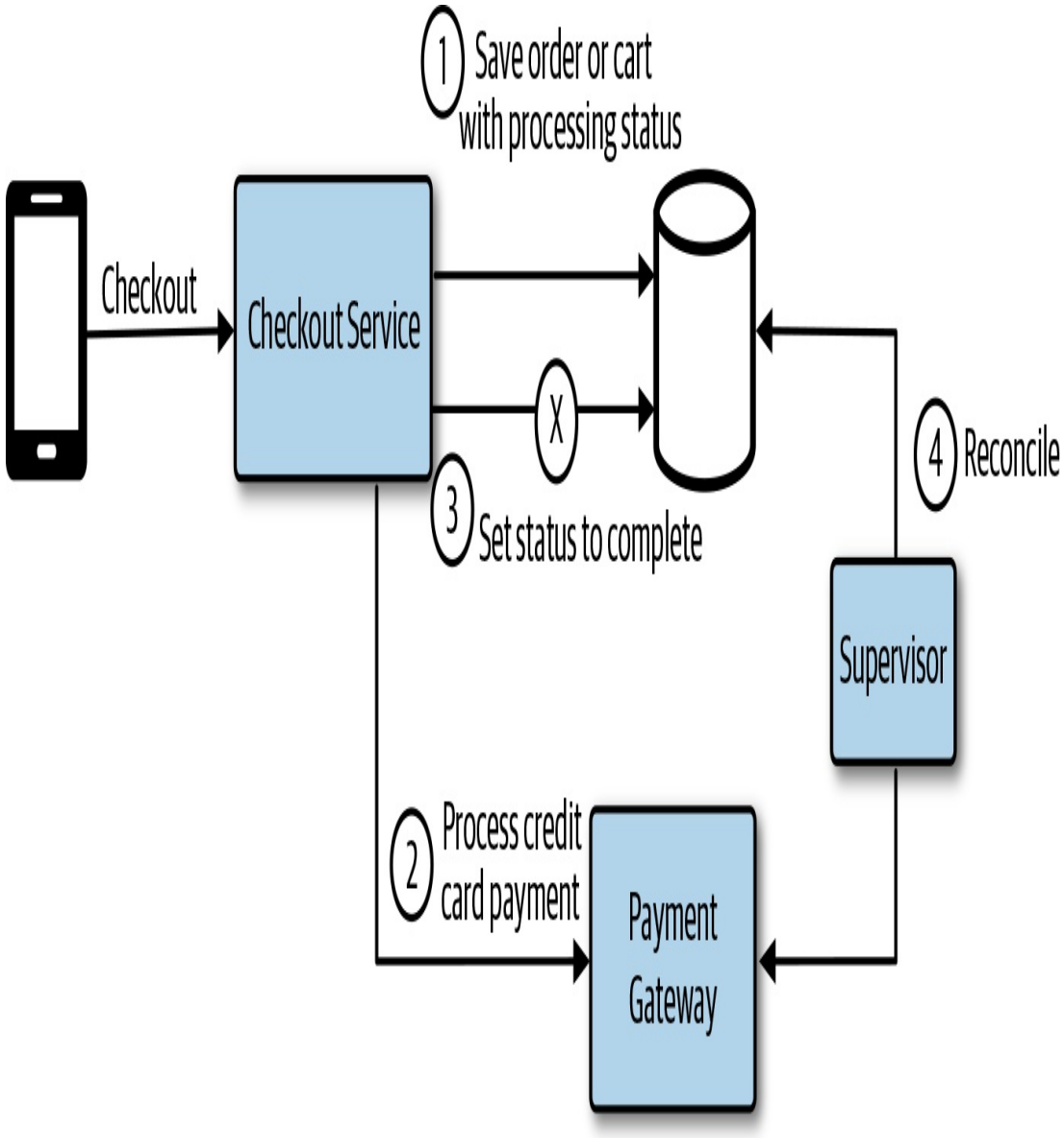

Figure 4-7. A supervisor service monitors transactions for errors

You can use this approach—using a supervisor and setting status—in many different ways to monitor systems and databases for consistency and take action to correct them or generate a notification of the issue.

Compensating Transactions

Traditional distributed transactions are not commonly used in today's cloud native applications, and not always available. There are situations for which transactions are necessary to maintain consistency across services or datastores. For example, a consumer posts some data with a file to an API requiring the application to write the file to object storage and some data to a document database. If we write the file to object storage and then fail when writing to the database, for any reason, we have a potentially orphaned file in object storage if the only way to find it is through a query on the database and reference. This is a situation in which we want to treat writing the file and the database record as a transaction; if one fails, both should fail. The file then should be removed to compensate for the failed database write. This is essentially what a compensating transaction does. A logical set of operations need to complete; if one of the operations fails, we might need to compensate the ones that succeeded.

> **NOTE**
>
> You should avoid service coordination. In many cases, you can avoid complex transaction coordination by designing for eventual consistency and using techniques like CDC.

Extract, Transform, and Load

The need to move and transform data for business intelligence (BI) is quite common. Businesses have been using Extract, Transform, and Load (ETL) platforms for a long time to move data from one system to another. Data analytics is becoming an important part of every business, large and small, so it should be no surprise that ETL platforms have become increasingly important. Data has become spread out across more systems and analytics

tools have become much more accessible. Everyone can take advantage of data analytics, and there's a growing need to move the data into a location for performing data analysis, like a data lake or date warehouse. You can use ETL to get the data from these operational data systems into a system to be analyzed. ETL is a process that comprises the following three different stages:

Extract

: Data is extracted or exported from business systems and data storage systems, legacy systems, operational databases, external services, and event Enterprise Resource Planning (ERP) or Customer Relationship Management (CRM) systems. When extracting data from the various sources, it's important to determine the velocity, how often the data is extracted from each source, and the priority across the various sources.

Transform

: Next, the extracted data is transformed; this would typically involve a number of data cleansing, transformation, and enrichment tasks. The data can be processed off a stream and is often stored in an interim staging store for batch processing.

Load

: The transformed data then is loaded into the destination and can be analyzed for BI.

All of the major cloud providers offer managed ETL services, like AWS Glue, Azure Data Factory, and Google Cloud DataFlow. Moving and processing data from one source to another is increasingly important and common in today's cloud native applications.

Microservices and Data Lakes

One challenge of dealing with decentralized data in a microservices

architecture is the need to perform reporting or analysis across data in multiple services. Some reporting and analytics requirements will need the data from the services to be in a common datastore.

> **NOTE**
>
> It might not be necessary to move the data in order to perform the required analysis and reporting across all of the data. Some or all of the analysis can be performed on each of the individual datastores in conjunction with some centralized analysis tasks on the results.

Having each service work from a shared or common database will, however, violate one of the microservices principles and potentially introduce coupling between the services. A common way to approach this is through data movement and aggregating the data into a location for a reporting or analytics team. In Figure 4-8, data from multiple microservices datastores is aggregated into a centralized database in order to deliver the necessary reporting and analytics requirements.

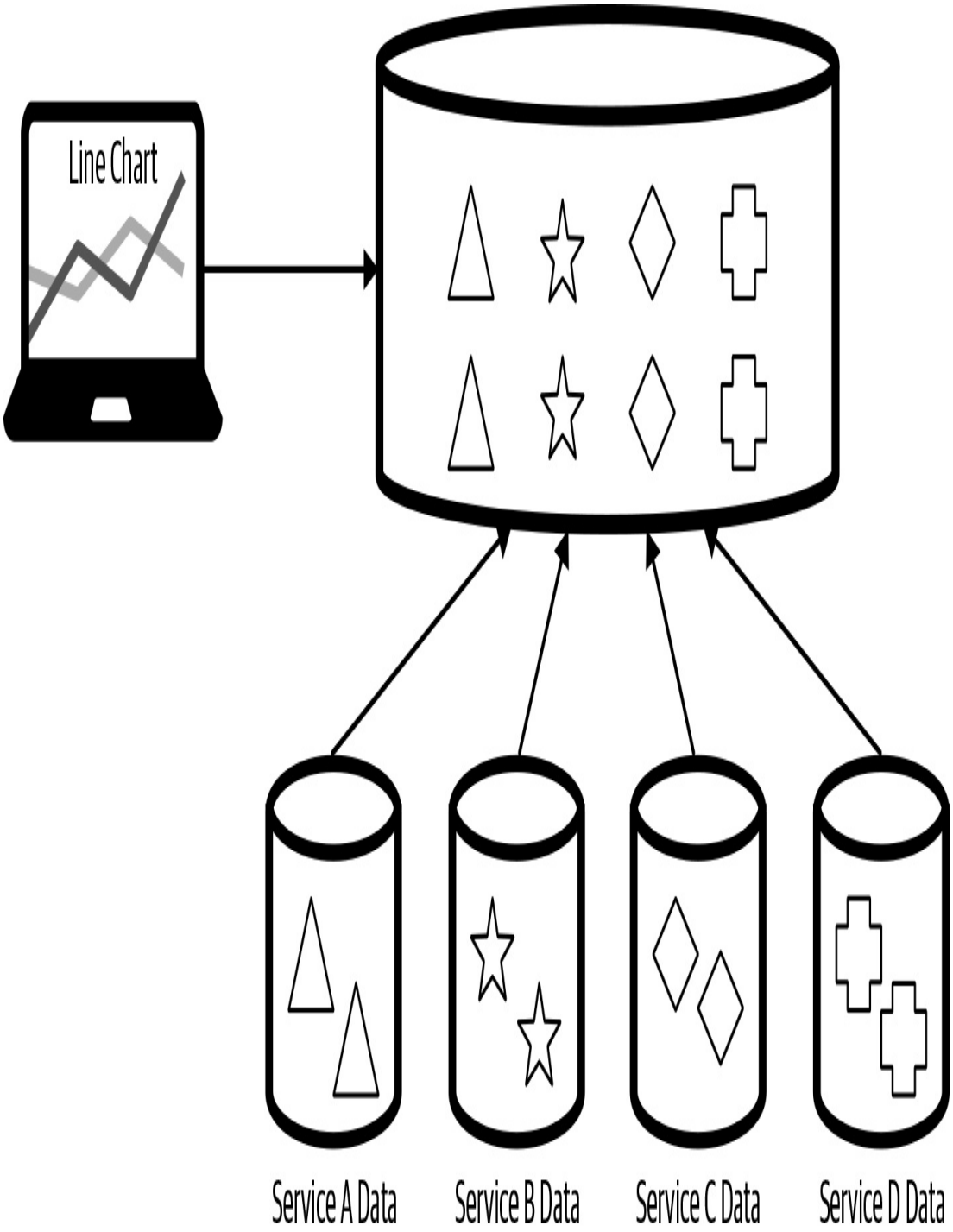

Figure 4-8. Data from multiple microservices aggregated in a centralized datastore

The data analytics or reporting team will need to determine how to get the data from the various service teams that it requires for the purpose of reporting without introducing coupling. There are a number of ways to

approach this, and it will be important to ensure loose coupling is maintained, allowing the teams to remain agile and deliver value quickly.

The individual services team could give the data analytics teams read access to the database and allow them to replicate the data, as depicted in Figure 4-9. This would be a very quick and easy approach, but the service team does not control when or how much load the data extraction will put on the store, causing potential performance issues. This also introduces coupling, and it's likely that the service teams then will need to coordinate with the data analytics team when making internal schema changes. The ETL load on the database adversely affecting service performance can be addressed by giving the data analytics team access to a read replica instead of the primary data. It might also be possible to give the data analytics team access to a view on the data instead of the raw documents or tables. This would help to mitigate some of the coupling concerns.

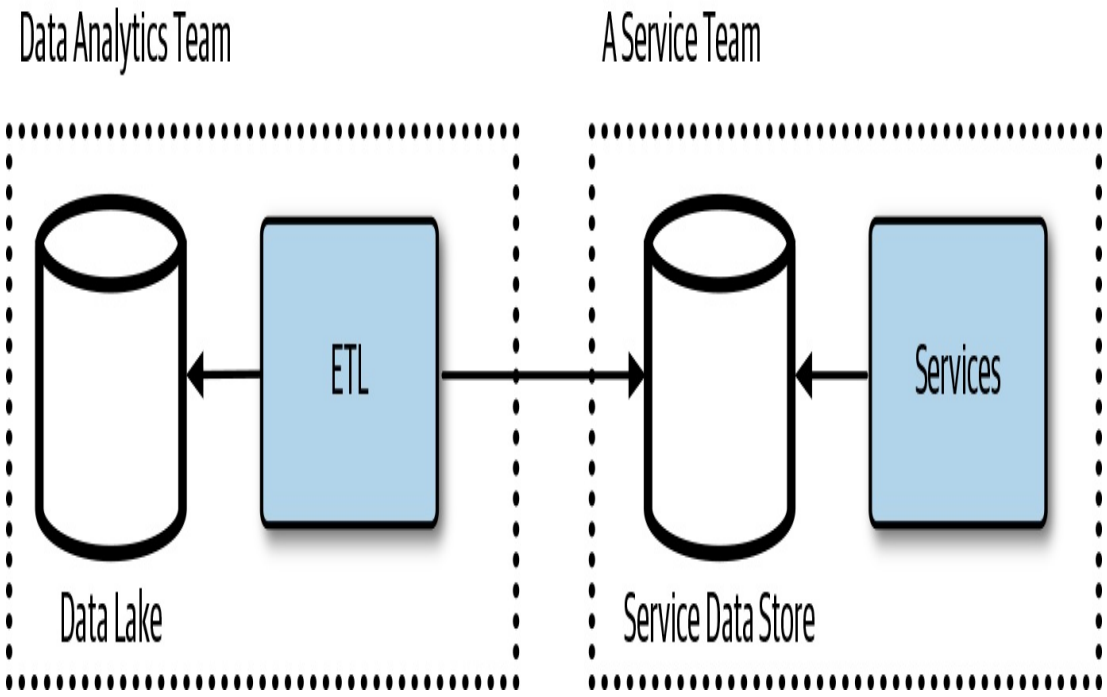

Figure 4-9. The data analytics team consumes data directly from the service team's database

This approach can work in the early phases of the application with a handful of services, but it will be challenging as the application and teams grow. Another approach is to use an *integration datastore*. The service team provisions and maintains a datastore for internal integrations, as shown in Figure 4-10. This allows the service team to control what data and the shape of the data in the integration repository. This integration repository should be managed like an API, documented and versioned. The service team could run ETL jobs to maintain the database or use CDC and treat it like a materialized view. The service team could make changes to its operational store without affecting the other teams. The service team would be responsible for the integration store.

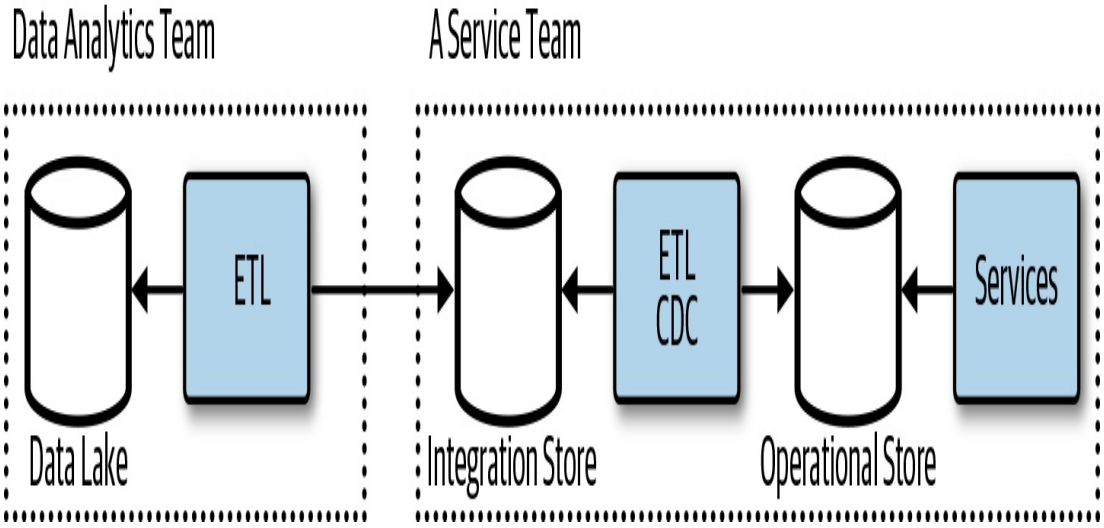

Figure 4-10. Database as an API

This could be turned around such that a service consumer, like the data analytics team, asks a service team to export or write data to the data lake, as illustrated in Figure 4-11, or to a staging store, as in Figure 4-12. The service teams support replication or data, logs, or data exports to a client-provided location as part of the service features and API. The data analytics team would provision a store or location in a datastore for each service team. The data analytics team then subscribes to data needed for

aggregated analytics.

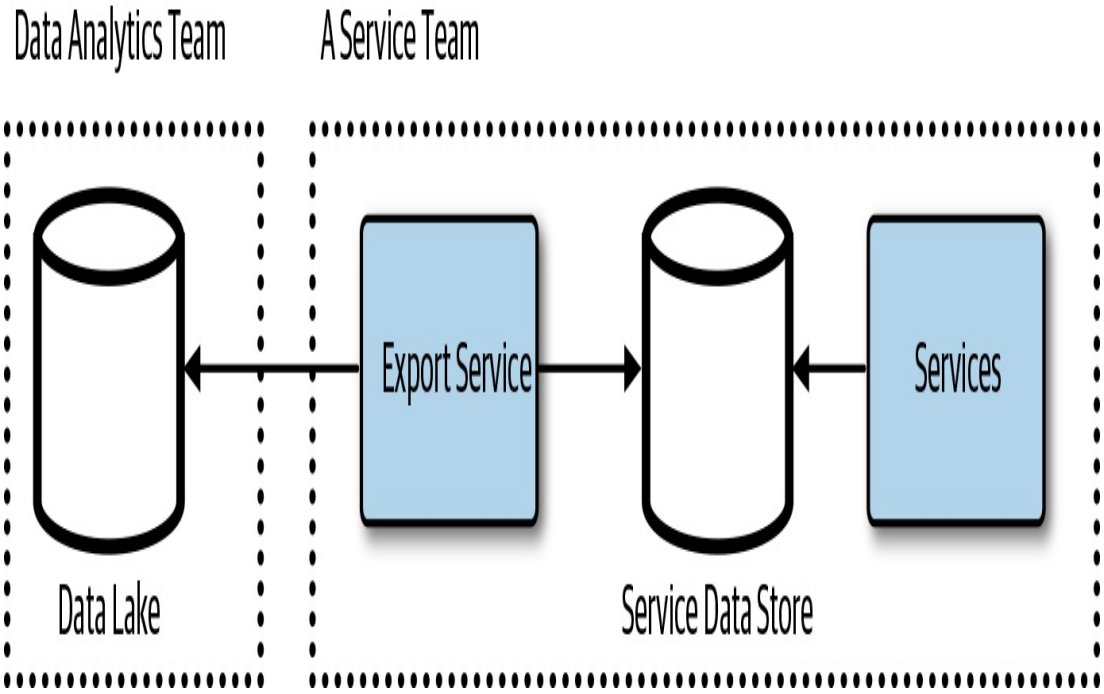

Figure 4-11. Service team data export service API

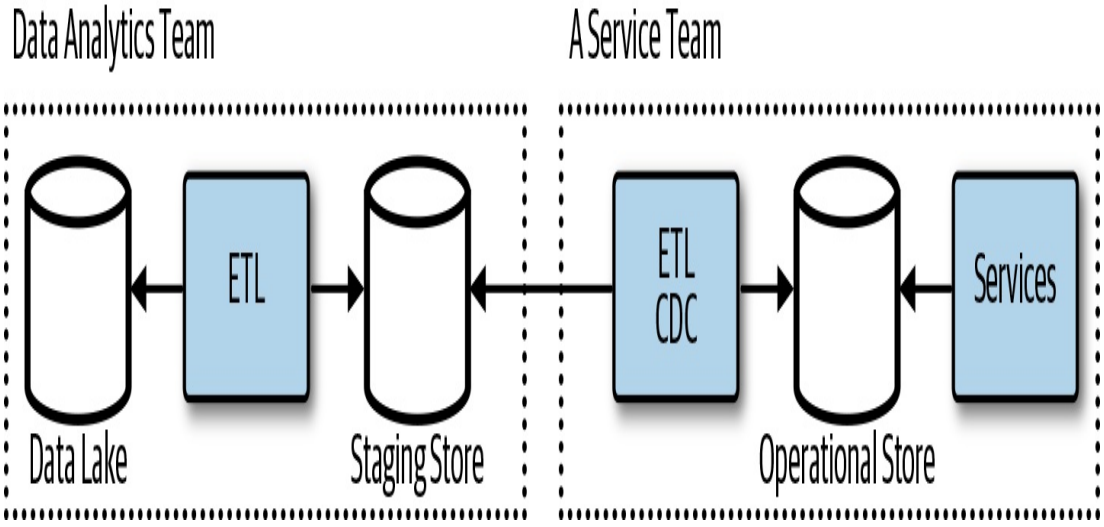

Figure 4-12. Service teams write to a staging store

It’s not uncommon for services to support data exports. The service implementation would define what export format and protocols are part of its API. This, for example, would be a configuration for an object storage location and credentials to which to send nightly exports, or maybe a

webhook to which to send batches of changes. A service consumer such as the data analytics team would have access to the service API, allowing it to subscribe to data changes or exports. The team could send locations and credentials to which to either dump export files or send events.

Client Access to Data

Clients applications generally do not have direct access to the datastores in most applications built today. Data is commonly accessed through a service that's responsible for performing authorizations, auditing, validation, and transformation of the data. The service is usually responsible for carrying out other functions, although in many data-centric applications, a large part of the service implementation simply handles data read and write operations.

A simple data-centric application would generally require you to build and operate a service that performs authentication, authorization, logging, transformations, and validation of data. It does, however, need to control who can access what within the datastore and validate what's being written. Figure 4-13 shows a typical frontend application calling a backend service that reads and writes to a single database. This is a common architecture for many applications today.

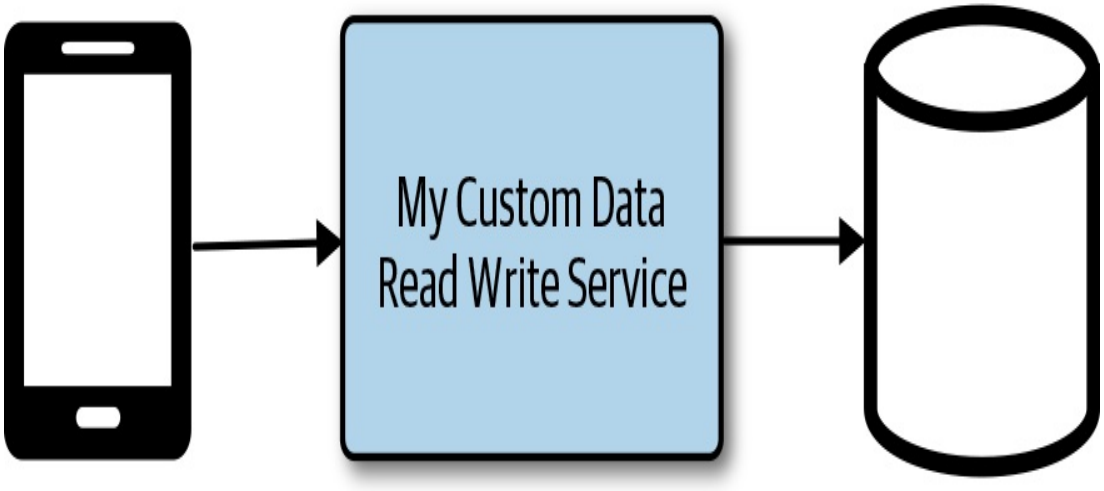

Figure 4-13. Client application with a backend service and database

Restricted Client Tokens (Valet-Key)

A service can create and return a token to a consumer that has limited use. This can actually be implemented using OAuth or even a custom cryptographically signed policy. The valet key is commonly used as a metaphor to explain how OAuth works and is a commonly used cloud design pattern. The token returned might be able to access only a specific data item for a limited period of time or upload a file to a specific location in a datastore. This can be a convenient way to offload processing from a service, reducing the cost and scale of the service and delivering better performance. In Figure 4-14, a file is uploaded to a service that writes the file to storage.

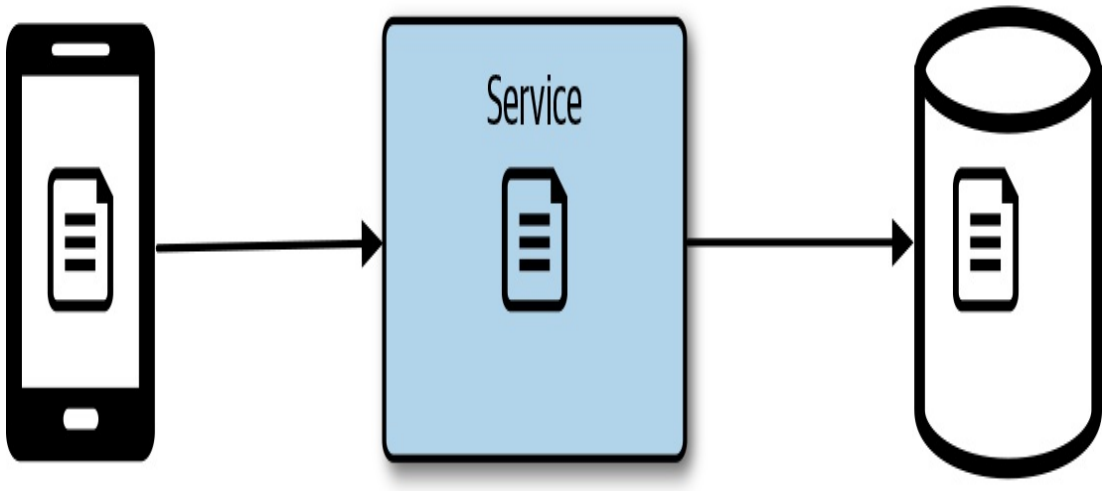

Figure 4-14. Client uploading a file that's passed through the service

Instead of streaming a file through the service, it can be much more efficient to return a token to the client with a location to access the file if it were reading or uploading the file to a specific location. In Figure 4-15, the client requests a token and a location from the service, which then generates a token with some policies. The token policy can restrict the location to which the file can be uploaded, and it's a best practice to set an expiration so that the token cannot be used anytime later on. The token should follow the principle of *least privilege*, granting the minimum permissions necessary to complete the task. In Microsoft Azure Blob Storage, the token is also referred to as a *shared-access signature*, and in Amazon S3, this would be a *presigned URL*. After the file is uploaded, an object storage function could be used to update the application state.

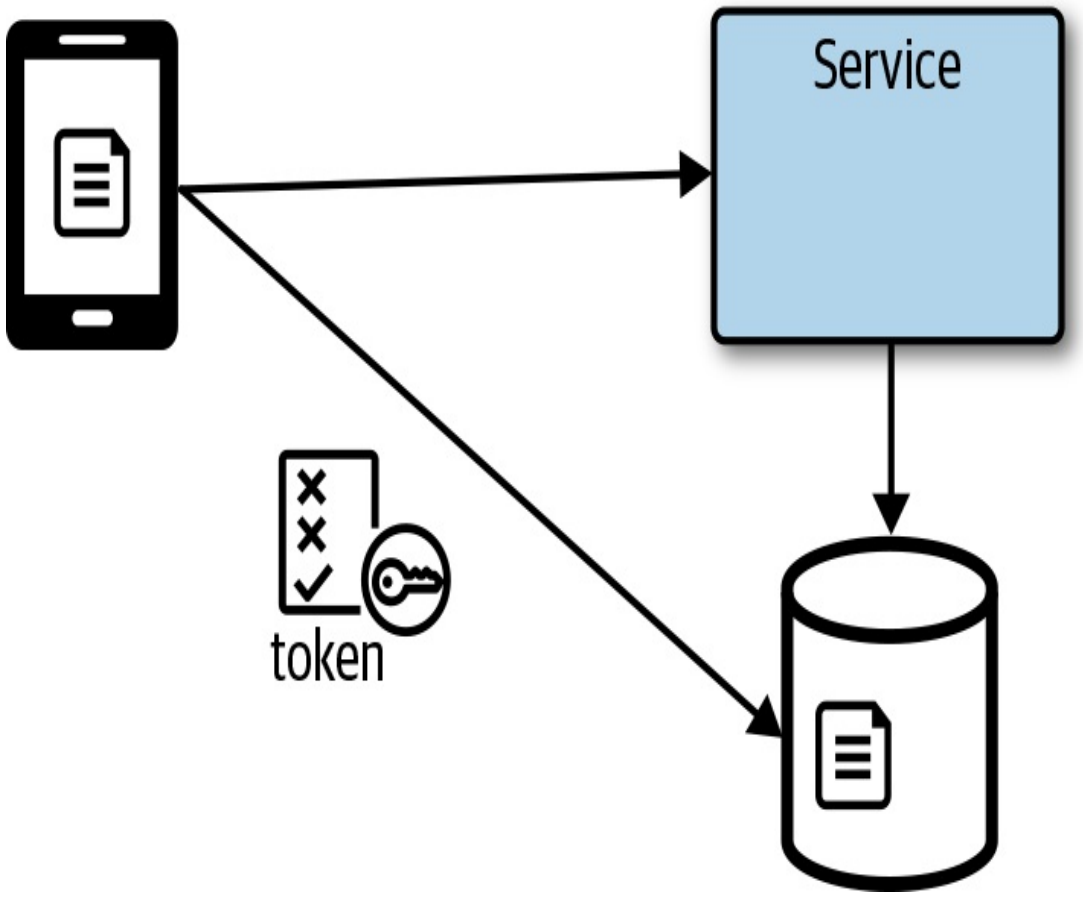

Figure 4-15. The client gets a token and path from a service to upload directly to storage

Database Services with Fine-Grained Access Control

Some databases provide fine-grained access control to data in the database. These database services are sometimes called a Backend as a Service (BaaS) or Mobile Backend as a Service (MBaaS). A full-featured MBaaS will generally offer more than just data storage, given that mobile applications often need identity management and notification services as well. This almost feels like we have circled back to the days of the old thick-client applications. Thankfully, data storage services have evolved so that it's not exactly the same. Figure 4-16 presents a mobile client connecting to a database service without having to deploy and manage an additional API. If there's no need to ship a customer API, this can be a

great way to quickly get an application out with low operational overhead. Careful attention is needed with releasing updates and testing the security rules to ensure that only the appropriate people are able to access the data.

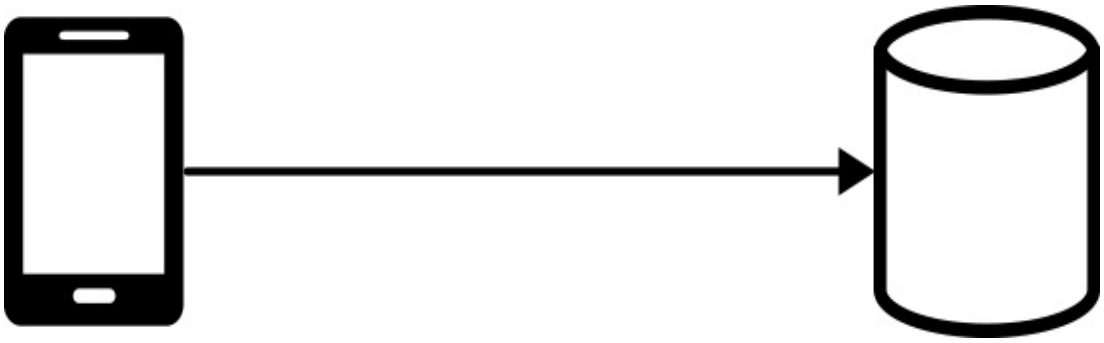

Figure 4-16. A mobile application connecting to a database

Databases such as Google's Cloud FireStore allow you to apply security rules that provide access control and data validation. Instead of building a service to control access and validate requests, you write security rules and validation. A user is required to authenticate to Google Firebase Authentication service, which can federate to other identity providers, like Microsoft's Azure Active Directory services. After a user is authenticated, the client application can connect directly to the database service and read or write data, provided the operations satisfy the defined security rules.

GraphQL Data Service

Instead of building and operating a custom service to manage client access to data, you can deploy and configure a GraphQL server to provide clients access to data. In Figure 4-17, a GraphQL service is deployed and configured to handle authorization, validation, caching, and pagination of data. Fully managed GraphQL services, like AWS AppSync, make it extremely easy to deploy a GraphQL-based backend for your client services.

> **NOTE**
>
> GraphQL is neither a database query language nor storage model; it's an API that returns application data based on a schema that's completely independent of how the data is stored.

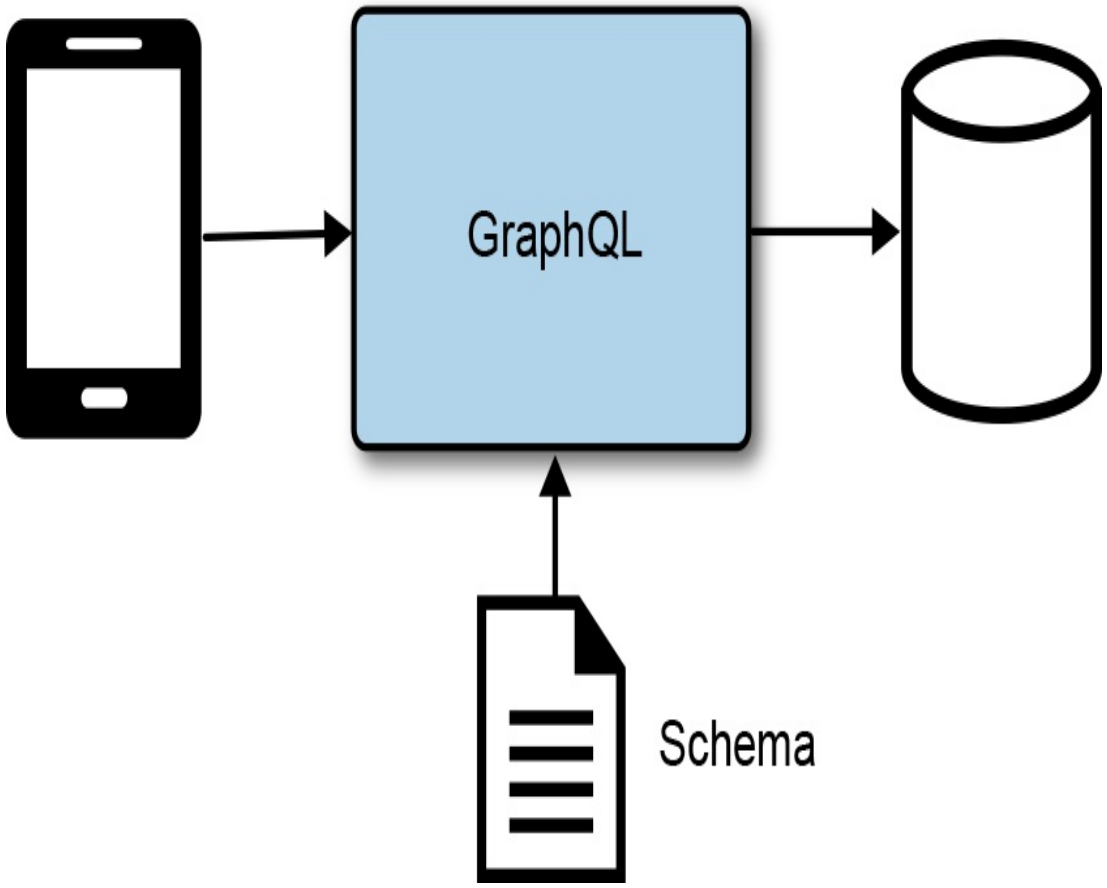

Figure 4-17. GraphQL data access service

GraphQL is flexible and configurable through a GraphQL specification. You can configure it with multiple providers, and even configure it to execute services either running in a container or deployed as functions that are invoked on request, as shown in Figure 4-18. GraphQL is a great fit for data-centric backends with the occasional service method that needs to be invoked. Services like GitHub are actually moving their entire API over to

GraphQL because this provides more flexibility to the consumers of the API. GraphQL can be helpful in addressing the over-fetching and chattiness that’s sometimes common with REST-based APIs.

GraphQL uses a schema-first approach, defining nodes (objects) and edges (relationships) as part of a schema definition for the graph structure. Consumers can query the schema for details about the types and relationships across the objects. One benefit of GraphQL is that it makes it easy to define the data you want, and only the data you want, without having to make multiple calls or fetch data that’s not needed. The specification supports authorizations, pagination, caching, and more. This can make it quick and easy to create a backend that handles most of the features needed in a data-centric application. For more information, visit the GraphQL website.

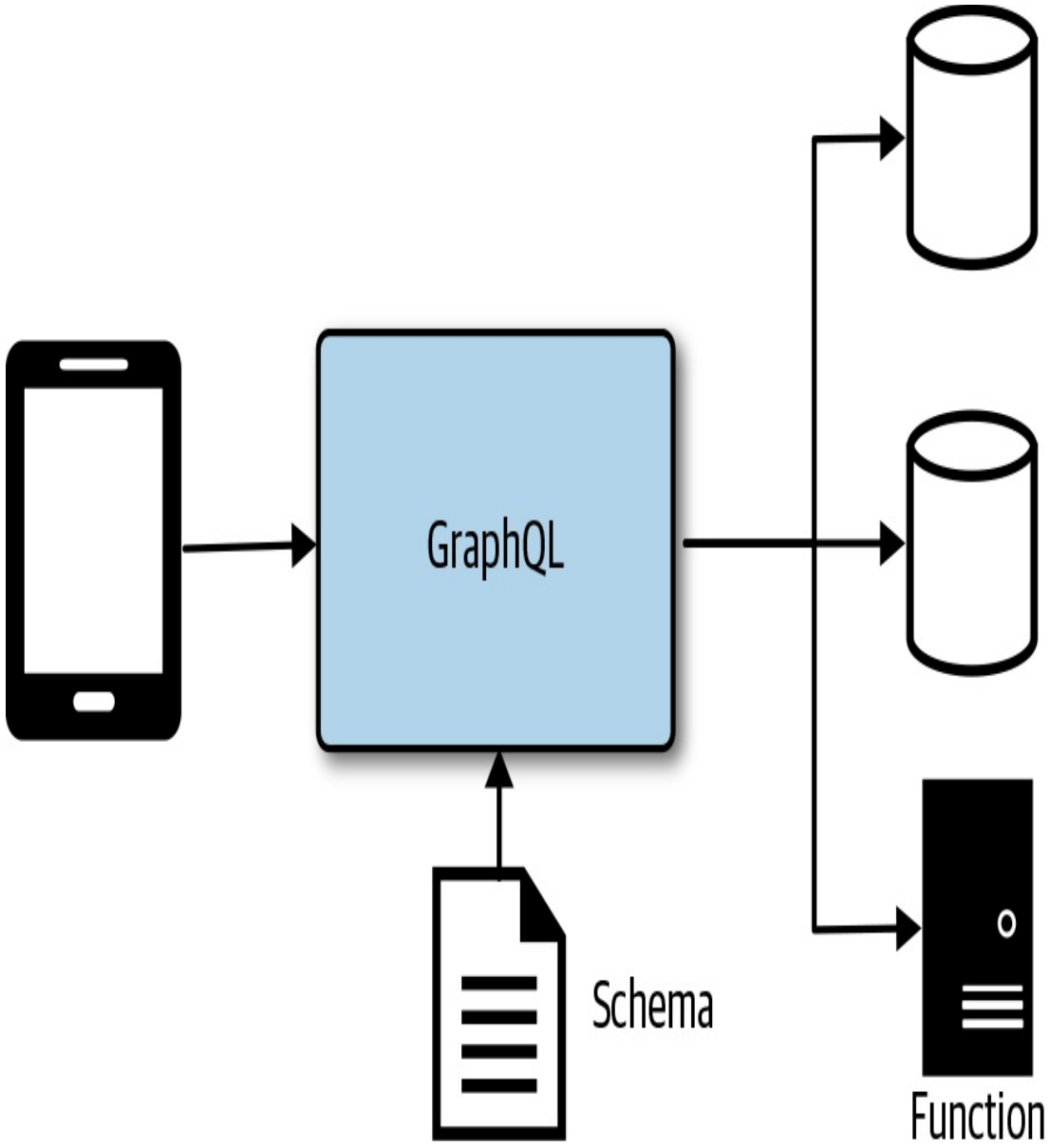

Figure 4-18. GraphQL service with multiple providers and execution

Fast Scalable Data

A large majority of application scaling and performance problems can be attributed to the databases. This is a common point of contention that can be challenging to scale out while meeting an application's data-quality requirements. In the past, it was too easy to put logic into a database in the form of stored procedures and triggers, increasing compute requirements

on a system that was notoriously expensive to scale. We learned to do more in the application and rely less on the database for something other than focusing on storing data.

> **TIP**
>
> There are very few reasons to put logic in a database. Don't do it. If you go there, make sure that you understand the trade-offs. It might make sense in a few cases and it might improve performance, but likely at the cost of scalability.

Scaling anything and everything can be achieved through replication and partitioning. Replicating the data to a cache, materialized view, or read-replica can help increase the scalability, availability, and performance of data systems. Partitioning data either horizontally through sharding, vertically based on data model, or functionally based on features will help improve scalability by distributing the load across systems.

Sharding Data

Sharding data is about dividing the datastore into horizontal partitions, known as *shards*. Each shard contains the same schema, but holds a subset of the data. Sharding often is used to scale a system by distributing the load across multiple data storage systems.

When sharding data, it's important to determine how many shards to use and how to distribute the data across the shards. Deciding how to distribute the data across shards heavily depends on the application's data. It's important to distribute the data in such a way that one single shard does not become overloaded and receive all or most of the load. Because the data for each shard or partition is commonly in a separate datastore,

it’s important that the application can connect to the appropriate shard (partition or database).

Caching Data

Data caching is important to scaling applications and improving performance. Caching is really just about copying the data to a faster storage medium like memory, and generally closer to the consumer. There might even be varying layers of cache; for example, data can be cached in the memory of the client application and in a shared distributed cache on the backend.

When working with a cache, one of the biggest challenges is keeping the cached data synchronized with the source. When the source data changes, it is often necessary to either invalidate or update the cached copy of the data. Sometimes, the data rarely changes; in fact, in some cases the data will not change through the lifetime of the application process, making it possible to load this static data into a cache when the application starts and then not need to worry about invalidation. Here are some common approaches for cache invalidation and updates:

- Rely on TTL configurations by setting a value that removes a cached item after a configurable expiration time. The application or a service layer then would be responsible for reloading the data when it does not find an item in the cache.
- Use CDC to update or invalidate a cache. A process subscribes to a datastore change stream and is responsible for updating the cache.
- Application logic is responsible for invalidating or updating the cache when it makes changes to the source data.
- Use a passthrough caching layer that’s responsible for managing

cached data. This can remove the concern of the data caching implementation from the application.

- Run a background service at a configuration interval to update a cache.
- Use the data replication features of the database or another service to replicate the data to a cache.
- Caching layer renews cached items based on access and available cache resources.

Content Delivery Networks

A content delivery network (CDN) is a group of geographically distributed datacenters, also known as points of presence (POP). A CDN often is used to cache static content closer to consumers. This reduces the latency between the consumer and the content or data needed. Following are some common CDN use cases:

- Improve website loading times by placing content closer to the consumer.
- Improve application performance of an API by terminating traffic closer to the consumer.
- Speed up software downloads and updates.
- Increase content availability and redundancy.
- Accelerate file upload through CDN services like Amazon CloudFront.

The content is cached, so a copy of it is stored at the edge locations and will be used instead of the source content. In Figure 4-19, a client is fetching a file from a nearby CDN with a much lower latency of 15 ms as opposed to the 82 ms latency between the client and the source location of

the file, also known as the *origin*. Caching and CDN technologies enable faster retrieval of the content, and scale by removing load from the origin as well.

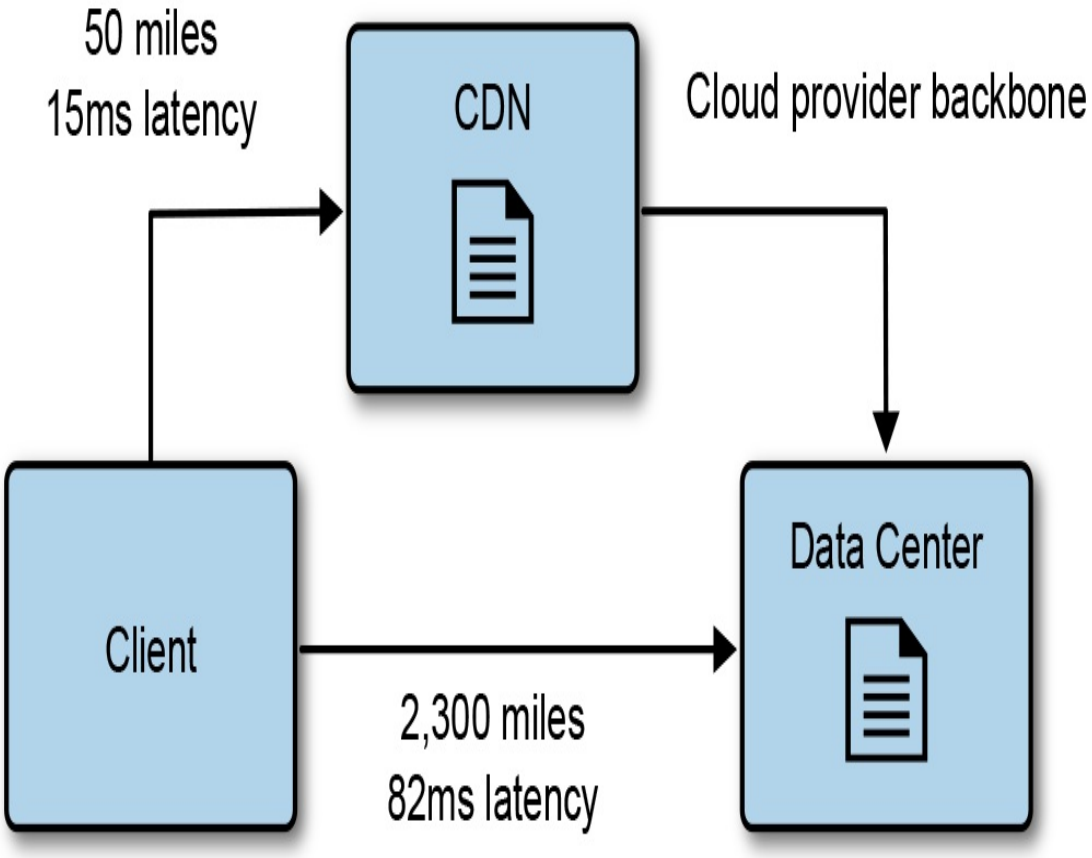

Figure 4-19. A client accesses content cached in a CDN closer to the client

The content cached in a CDN is usually configured with an expiration date-time, also known as *TTL properties*. When the expiration date-time is exceeded, the CDN reloads the content from the origin, or source. Many CDN services allow you to explicitly invalidate content based on a path; for example, */img/**. Another common technique is to change the name of the content by adding a small hash to it and updating the reference for consumers. This technique is commonly used for web application bundles like the JavaScript and CSS files used in a web application.

Here are some considerations regarding CDN cache management:

- Use content expiration to refresh content at specific intervals.
- Change the name of the resource by appending a hash or version to the content.
- Explicitly expire the cache either through management console or API.

CDN vendors continue adding more features, making it possible to push more and more content, data, and services closer to the consumers, improving performance, scale, security, and availability. Figure 4-20 demonstrates a client calling a backend API with the request being routed through the CDN and over the cloud provider's backbone connection between datacenters. This is a much faster route to the API with lower latency, improving the Secure Sockets Layer (SSL) handshake between the client and the CDN as well as the API request.

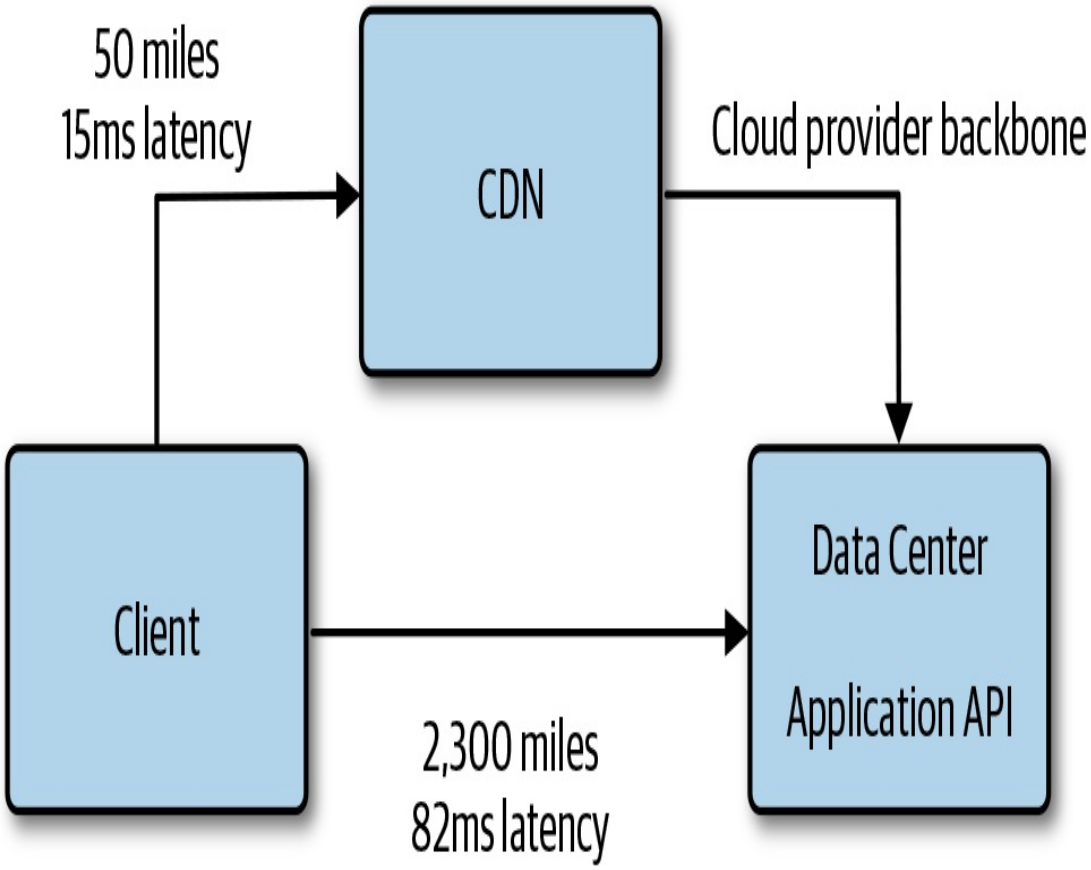

Figure 4-20. Accelerated access to a backend API

Here are a few additional features to consider when using CDN technologies:

Rules or behaviors

: It can be necessary to configure routing, adding response headers, or enable redirects based on request properties like SSL.

Application logic

: Some CDN vendors like Amazon CloudFront allow you to run application logic at the edge, making it possible to personalize content for a consumer.

Custom name

: It’s often necessary to use a custom name with SSL, especially when

serving a website through a CDN.

File upload acceleration

: Some CDN technologies are able to accelerate file upload by reducing the latency to the consumer.

API acceleration

: As with file upload, it's possible to accelerate APIs through a CDN by reducing the latency to the consumer.

> **NOTE**
>
> Use a CDN as much as possible, pushing as much as you can over the CDN.

Analyzing Data

The data created and stored continues to grow at exponential rates. The tools and technologies used to extract information from data continues to evolve to support the growing demand to derive insights from the data, making business insights through complex analytics available to even the smallest businesses.

Streams

Businesses need to reduce their time to insights in order to gain an edge in today's competitive fast-moving markets. Analyzing the data streams in real time is a great way to reduce this latency. Streaming data-processing engines are designed for unbounded datasets. Unlike data in a traditional data storage system in which you have a holistic view of the data at a specific point in time, streams have an entity-by-entity view of the data over time. Some data, like stock market trades, click streams, or sensor

data from devices, comes in as a stream of events that never end. Stream processing can be used to detect patterns, identify sequences, and look at results. Some events, like a sudden transition in a sensor, might be more valuable when they happen and diminish over time or enable a business to react more quickly and immediately to these important changes. Detecting a sudden drop in inventory, for example, allows a company to order more stock and avoid some missed sales opportunities.

Batch

Unlike stream processing, which is done in real time as the data arrives, batch processing is generally performed on very large bounded sets of data as part of exploring a data science hypothesis, or at specific intervals to derive business insights. Batch processing is able to process all or most of the data and can take minutes or hours to complete, whereas stream processing is completed in a matter of seconds or less. Batch processing works well with very large volumes of data, which might have been stored over a long period of time. This could be data from legacy systems or simply data for which you’re looking for patterns over many months or years.

Data analytics systems typically use a combination of batch and stream processing. The approaches to processing streams and batches have been captured as some well-known architecture patterns. The Lambda architecture is an approach in which applications write data to an immutable stream. Multiple consumers read data from the stream independent of one another. One consumer is concerned with processing data very quickly, in near real time, whereas the other consumer is concerned with processing in batch and a lower velocity across a larger set of data or archiving the data to object storage.

Data Lakes on Object Storage

Data lakes are large, scalable, and generally centralized datastores that allow you to store structured and unstructured data. They are commonly used to run map-and-reduce jobs for analyzing vast amounts of data. The analytics jobs are highly parallelizable so the analysis of the data can easily be distributed across the store. Hadoop has become the popular tool for data lakes and big data analysis. Data is commonly stored on a cluster of computers in the Hadoop Distributed File System (HDFS), and various tools in the Hadoop ecosystem are used to analyze the data. All of the major public cloud vendors provide managed Hadoop clusters for storing and analyzing the data. The clusters can become expensive, requiring a large number of very big machines. These machines might be running even when there are no jobs to run on the cluster. It is possible to shut down these clusters and maintain state for cost savings when they are not in use and resume the clusters during periods of data loading or analysis.

It’s becoming increasingly common to use fully managed services that allow you to pay for the data loaded in the service and pay-per-job execution. These services not only can reduce operational costs related to managing these services, but also can result in big savings when running the occasional analytics jobs. Cloud vendors have started providing services that align with a serverless cost model for provisioning data lakes. Azure Data Lake and Amazon S3–based AWS Lake Formation are some examples of this.

Data Lakes and Data Warehouses

Data lakes are often compared and contrasted with data warehouses because they are similar, although in large organizations it’s not uncommon to see both used. Data lakes are generally used to store raw

and unstructured data, whereas the data in a data warehouse has been processed and organized into a well-defined schema. It's common to write data into a data lake and then process it from the data lake into a data warehouse. Data scientists are able to explore and analyze the data to discover trends that can help define what is processed into a data warehouse for business professionals.

Distributed Query Engines

Distributed query engines are becoming increasingly popular, supporting the need to quickly analyze data stored across multiple data systems. Distributed query engines separate the query engine from the storage engine and use techniques to distribute the query across a pool of workers. A number of open source query engines have become popular in the market: Presto, Spark SQL, Drill, and Impala, to name a few. These query engines utilize a provider model to access various data storage systems and partitions.

Hadoop jobs were designed for processing large amounts of data through jobs that would run for minutes or even hours crunching through the vast amounts of data. Although a structured query language (SQL)–like interface exists in tools such as HIVE, the queries are translated to jobs submitted to a job queue and scheduled. A client would not expect that the results from a job would return in minutes or seconds. It is, however, expected that distributed query engines like Facebook's Presto would return results from a query in the matter of minutes or even seconds.

At a high level, a client submits a query to the distributed query engine. A coordinator is responsible for parsing the query and scheduling work to a pool of workers. The pool of workers then connects to the datastores

needed to satisfy the query, fetches the results, and merges the results from each to the workers. The query can run against a combination of datastores: relational, document, object, file, and so on. Figure 4-21 depicts a query that fetches information from a MongoDB database and some comma-separated values (CSV) files stored in an object store like Amazon S3, Azure Blob Storage, or Google Object Storage.

The cloud makes it possible to quickly and easily scale workers, allowing the distributed query engine to handle query demands.

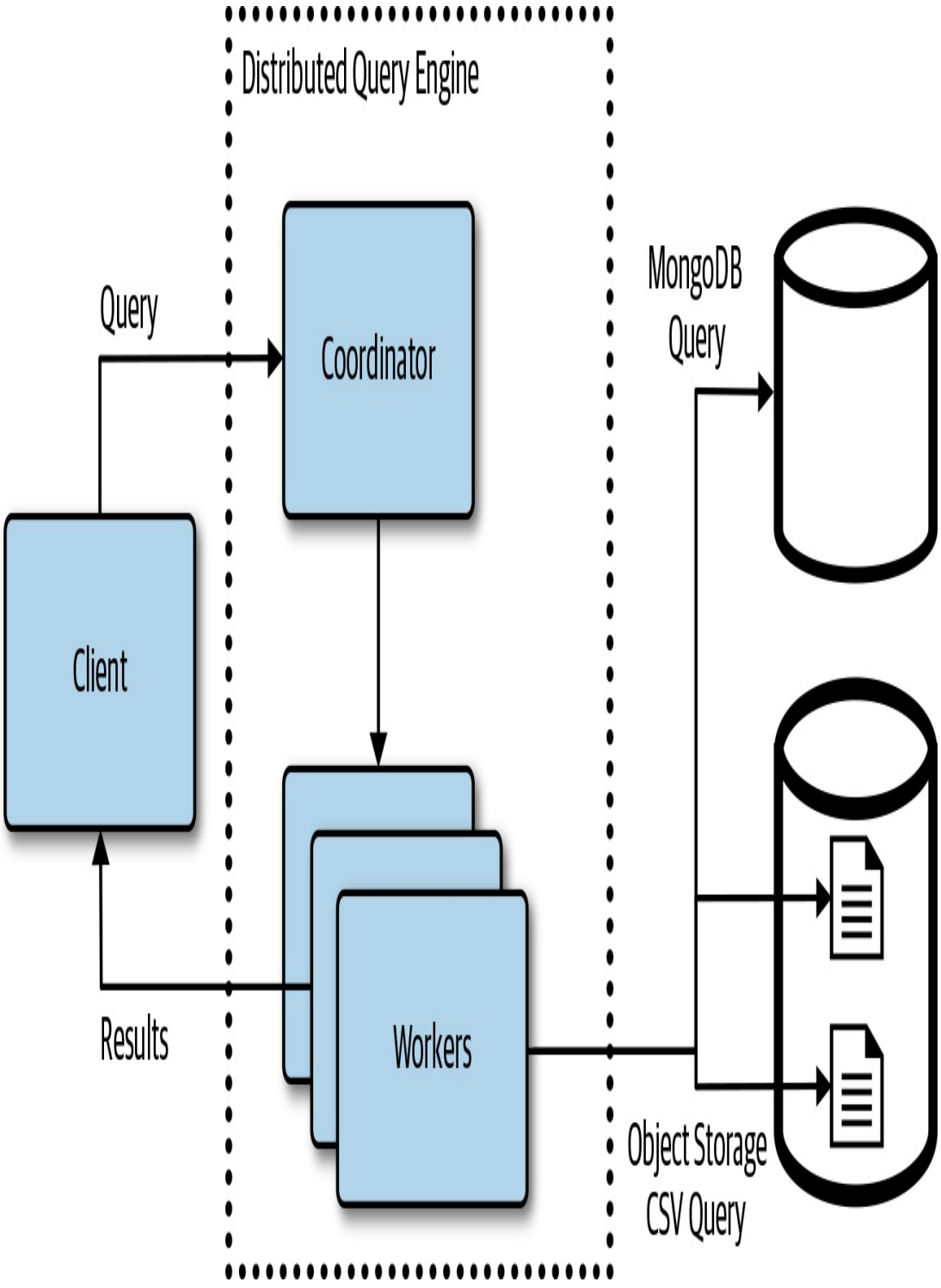

Figure 4-21. Overview of a distributed query engine

Databases on Kubernetes

Kubernetes dynamic environment can make it challenging to run data storage systems in a Kubernetes cluster. Kubernetes pods are created and destroyed, and cluster nodes can be added or removed, forcing pods to move to new nodes. Running a stateful workload like a database is much different than stateless services. Kubernetes has features like stateful sets and support for persistent volumes to help with deploying and operating databases in a Kubernetes cluster. Most of the durable data storage systems require a disk volume as the underlying persistent storage mechanism, so understanding how to attach storage to pods and how volumes work is important when deploying databases on Kubernetes.

In addition to providing the underlying storage volumes, data storage systems have different routing and connectivity needs as well as hardware, scheduling, and operational requirements. Some of the newer cloud native databases have been built for these more dynamic environments and can take advantage of the environments to scale out and tolerate transient errors.

> **NOTE**
>
> There are a growing number of operators available to help simplify the deployment and management of data systems on Kubernetes. Operator Hub is a directory listing of operators (*https://www.operatorhub.io*).

Storage Volumes

A database system like MongoDB runs in a container on Kubernetes and often needs a durable volume with a life cycle different from the container. Managing storage is much different than managing compute. Kubernetes volumes are mounted into pods using persistent volumes, persistent

volume claims, and underlying storage providers. Following are some fundamental storage volume terms and concepts:

Persistent volume

: A persistent volume is the Kubernetes resource that represents the actual physical storage service, like a cloud provider storage disk.

Persistent volume claim

: A persistent volume storage claim is a storage request, and Kubernetes will assign and associate a persistent volume to it.

Storage class

: A storage class defines storage properties for the dynamic provisioning of a persistent volume.

A cluster administrator will provision persistent volumes that capture the underlying implementation of the storage. This could be a persistent volume to a network-attached file share or cloud provider durable disks. When using cloud provider disks, it’s more likely one or more storage classes will be defined and dynamic provisioning will be used. The storage class will be created with a name that can be used to reference the resource, and the storage class will define a provisioner as well as the parameters to pass to the provisioner. Cloud providers offer multiple disk options with different price and performance characteristics. Different storage classes are often created with the different options that should be available in the cluster.

A pod is going to be created that requires a persistent storage volume so that data is still there when the pod is removed and comes back up on another node. Before creating the pod, a persistent volume claim is created, specifying the storage requirements for the workload. When a persistent volume claim is created, and references a specific storage class,

the provisioner and parameters defined in that storage class will be used to create a persistent volume that satisfies the persistent volume claims request. The pod that references the persistent volume claim is created and the volume is mounted at the path specified by the pod. Figure 4-22 shows a pod with a reference to a persistent volume claim that references a persistent volume. The persistent volume resource and plug-in contains the configuration and implementation necessary to attach the underlying storage implementation.

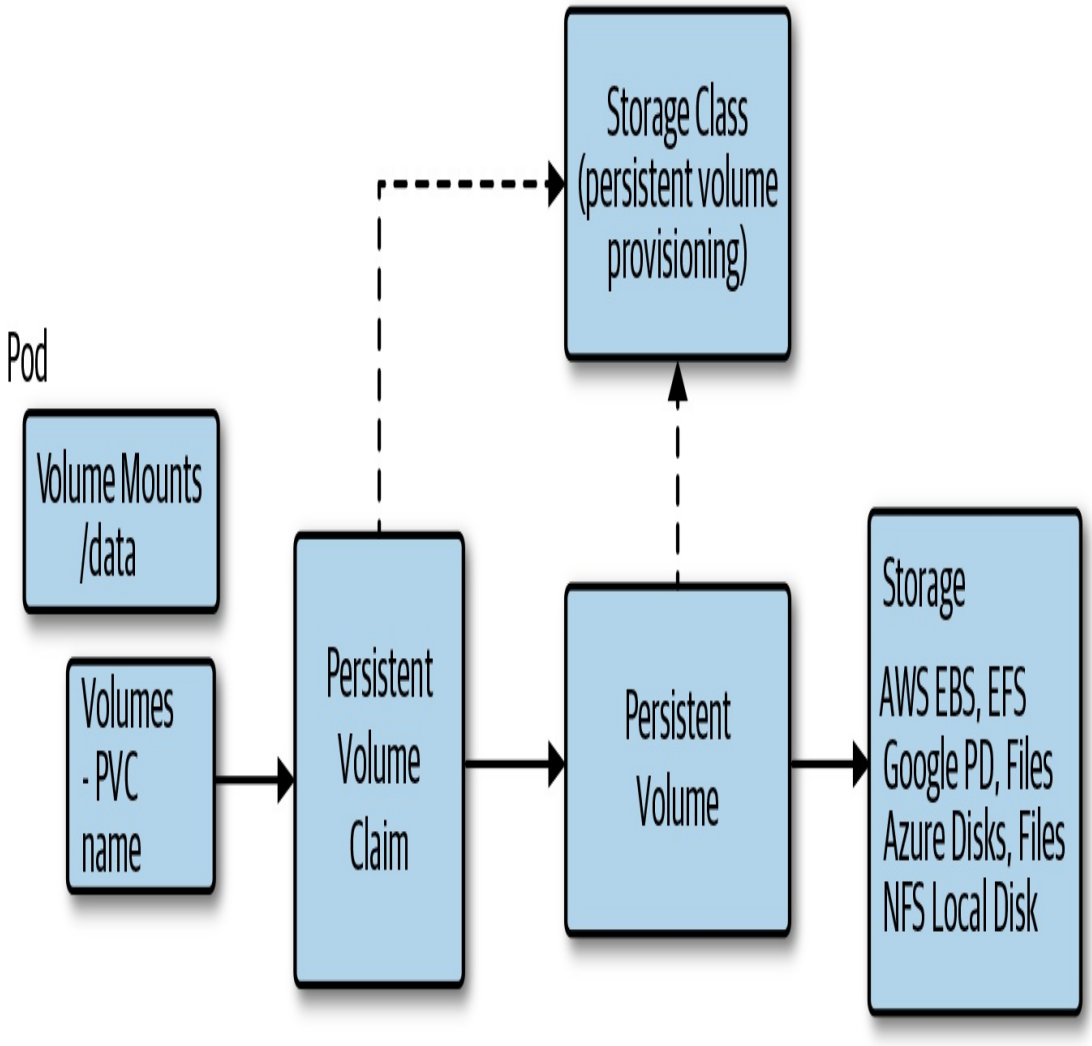

Figure 4-22. A Kubernetes pod persistent volume relationship

> **NOTE**
>
> Some data systems might be deployed in a cluster using ephemeral storage. Do not configure these systems to store data in the container; instead, use a persistent volume mapped to a node's ephemeral disks.

StatefulSets

StatefulSets were designed to address the problem of running stateful services like data storage systems on Kubernetes. StatefulSets manage the deployment and scaling of a set of pods based on a container specification. StatefulSets provide a guarantee about the order and uniqueness of the pods. The pods created from the specification each have a persistent identifier that is maintained across any rescheduling. The unique pod identity comprises the StatefulSet name and an ordinal starting with zero. So, a StatefulSet named "mongo" and a replica setting of "3" would create three pods named "mongo-0," "mongo-1," and "mongo-2," each of which could be addressed using this stable pod name. This is important because clients often need to be able to address a specific replica in a storage system and the replicas often need to communicate between one another. StatefulSets also create a persistent volume and persistent volume claim for each individual pod, and they are configured such that the disk created for the "mongo-0" pod is bound to the "mongo-0" pod when it's rescheduled.

> **NOTE**
>
> StatefulSets currently require a headless service, which is responsible for the network identity of the pods and must be created in addition to the StatefulSet.

Affinity and anti-affinity is a feature of Kubernetes that allows you to constrain which nodes pods will run on. Pod anti-affinity can be used to improve the availability of a data storage system running on Kubernetes by ensuring replicas are not running on the same node. If a primary and secondary were running on the same node and that node happened to go down, the database would be unavailable until the pods were rescheduled and started on another node.

Cloud providers offer many different types of compute instance types that are better suited for different types of workloads. Data storage systems will often run better on compute instances that are optimized for disk access, although some might require higher memory instances. The stateless services running the cluster, however, do not require these specialized instances that will often cost more and are fine running on general commodity instances. You can add a pool of storage-optimized nodes to a Kubernetes cluster to run the storage workloads that can benefit from these resources. You can use Kubernetes node selection along with taints and tolerations to ensure the data storage systems are scheduled on the pool of storage optimized nodes and that other services are not.

Given most data storage systems are not Kubernetes aware, it’s often necessary to create an adapter service that runs with the data storage system pod. These services are often responsible for injecting configuration or cluster environment settings into the data storage system. For example, if we deployed a MongoDB cluster and need to scale the cluster with another node, the MongoDB sidecar service would be responsible for adding the new MongoDB pod to the MongoDB cluster.

DaemonSets

A DaemonSet ensures that a group of nodes runs a single copy of a pod. This can be a useful approach to running data storage systems when the system needs to be part of the cluster and use nodes dedicated to storage system. A pool of nodes would be created in the cluster for the purpose of running the data storage system. A node selector would be used to ensure the data storage system was only scheduled to these dedicated nodes. Taints and tolerations would be used to ensure other processes were not scheduled on these nodes. Here are some trade-offs and considerations when deciding between daemon and stateful sets:

- Kubernetes StatefulSets work like any other Kubernetes pods, allowing them to be scheduled in the cluster as needed with available cluster resources.
- StatefulSets generally rely on remote network attached storage devices.
- DaemonSets offer a more natural abstraction for running on a database on a pool of dedicated nodes.
- Discovery and communications will add some challenges that need to be addressed.

Summary

Migrating and building applications in the cloud requires a different approach to the architecture and design of applications' data-related requirements. Cloud providers offer a rich set of managed data storage and analytics services, reducing the operating costs for data systems. This makes it much easier to consider running multiple and different types of data systems, using storage technologies that might be better suited for the task. This cost and scale of the datastores has changed, making it easier to store large amounts of data at a price point that keeps going down as cloud

providers continue to innovate and compete in these areas.

Chapter 5. DevOps

Developing, testing, and deploying cloud native applications differs significantly from traditional development and operations practices. In this chapter, you learn the fundamentals of DevOps along with the proven practices, including all of the benefits and challenges of developing, testing, and operating cloud native applications. Additionally, we cover designing cloud native applications with operations and rapid, reliable development processes in mind. Most concepts and patterns explained in this chapter are applicable to both containerized services and functions. When this is not the case, we explicitly call out the differences.

What Is DevOps?

DevOps is a broad concept that encompasses multiple aspects of collaboration and communication between software developers and other IT professionals. The easiest way to define DevOps is to talk about its goals. DevOps is intended to improve collaboration between development and operations teams throughout the entire process of software development, from planning to delivery, to improve deployment frequency, achieve faster time to market, lower the failure rate of new releases, shorten lead time between fixes, and improve mean time to recovery.

One of the models you can use when talking about DevOps is called CALMS, which stands for Collaboration, Automation, Lean, Measurement, and Sharing. The CALMS model is a method that we can

use to assess, analyze, and compare the maturity of the DevOps team.

Collaboration ...

Chapter 6. Best Practices

Throughout this book, you have learned about the fundamentals of cloud native applications—how to design, develop, and operate them as well as how to deal with data. To conclude, this chapter aims to provide a laundry list covering tips, proven techniques, and proven best practices to build and manage reactive cloud native applications.

Moving to Cloud Native

In Chapter 2, you learned about the process that many customers follow when moving traditional applications to the cloud. There are many best practices and lessons learned that you should consider when moving an existing application into the cloud.

Breaking Up the Monolith for the Right Reasons

"Never change a running system" is a widely used statement in software development, and it is also applicable when you consider moving your application to the cloud. If your sole requirement is to move your application to the cloud, you can always consider moving it on Infrastructure as a Service (IaaS)—in fact, that should be your very first step. That said, there are benefits of redesigning your application to be cloud native, but you need to weigh the pros and cons. Following are some guidelines indicating that a redesign makes sense:

- Your codebase has grown to a point that it takes very long to release an updated version and thus you cannot react to new

market or customer requirements quickly.

- Components of your applications have different scale requirements. A good example is a traditional three-tier application consisting of a frontend, business, and data tier. Only the frontend tier might experience heavy load in user requests, whereas the business and data tier are still comfortably handling the load. As mentioned in Chapter 2 and Chapter 3, cloud native applications allow you to scale services independently.
- Better technology choices have emerged. There is constant innovation in the technology sector, and some new technologies might be better suited for parts of your application.

After you have decided that you want to redesign your application, you need to consider many things. In the following sections, we provide a comprehensive look at these considerations.

Decouple Simple Services First

Start by breaking off components that provide simpler functionality because they usually do not have a lot of dependencies and, thus, are not deeply integrated within the monolith.

Learn to Operate on a Small Scale

Use the first service as a learning path for how to operate in a cloud native world. Starting with a simple service, you can focus on setting up automation to provision the infrastructure and the CI/CD pipeline so that you become familiar with the process of developing, deploying, and operating a cloud native service. Having a simple service and minimal infrastructure will allow you to learn, exercise, and improve your new process ahead of time, without substantial impact on the monolith and your end users.

Use an Anticorruption Layer Pattern

Nothing is perfect, especially in the software development world, so you will eventually end up with a new service that makes calls back to the monolith. In this case, you might want to use the *Anticorruption Layer* pattern. This pattern is used to implement a facade or adapter between components that don’t share the same semantics. The purpose of the anticorruption layer is to translate the request from one component to another; for example, implementing protocol or schema translations.

To implement this, you design and create a new API in the monolith that makes calls through the anticorruption layer in the new service, as shown in Figure 6-1.

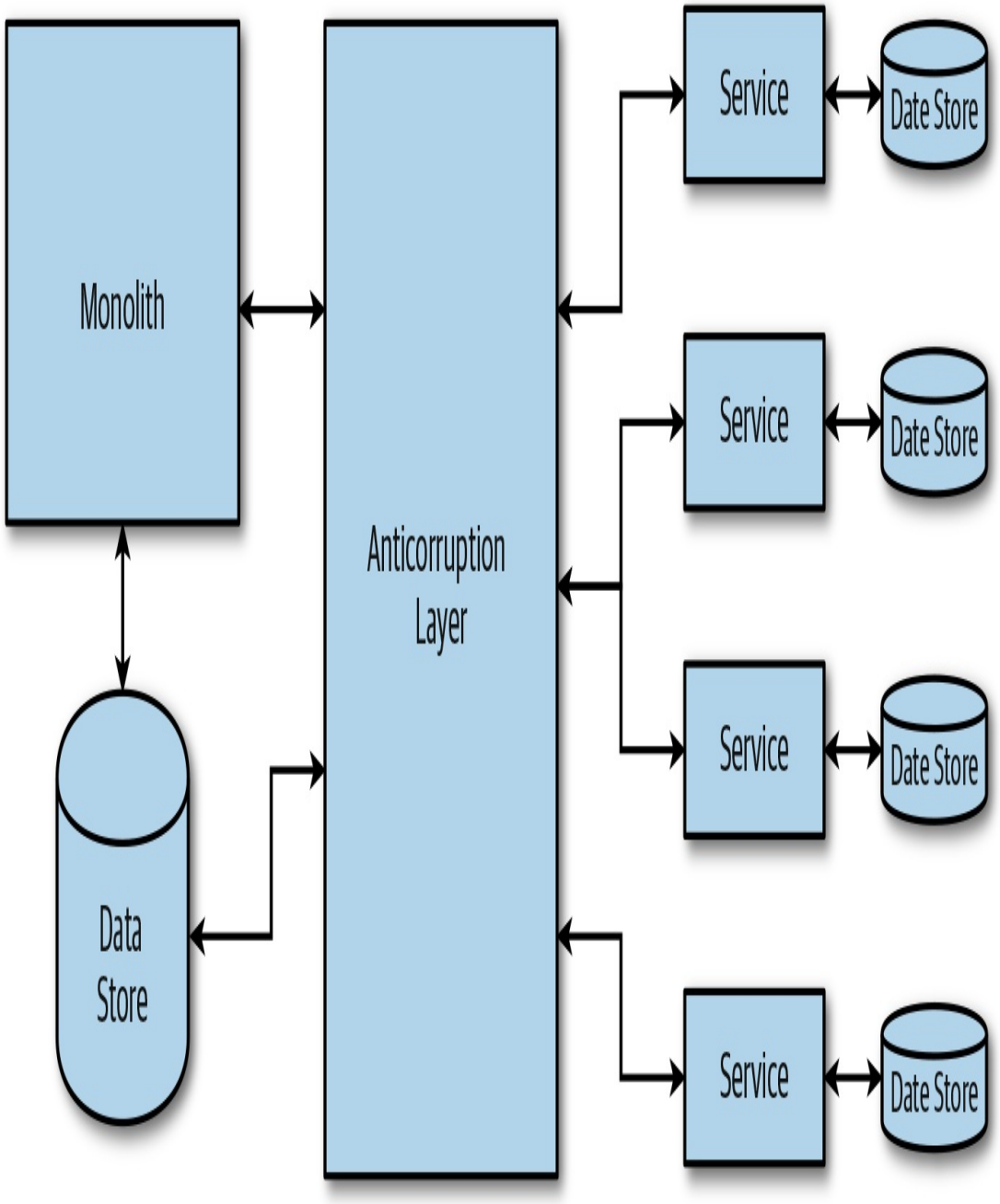

Figure 6-1. Anticorruption Layer pattern

There are a couple of considerations when you are using this approach. As Figure 6-1 illustrates, the anticorruption layer is a service on its own, so you need to think about how to scale and operate the layer. Also, you need to think about whether you want to retire the anticorruption layer after the monolithic application has been fully moved into a cloud native

application.

Use a Strangler Pattern

When you are decomposing your monolith to move to microservices and functions, you can use a gateway and a pattern such as a *Strangler* pattern. The idea behind the Strangler pattern is to use the gateway as a facade while you gradually move the backend monolith to a new architecture—either services, functions, or a combination of both. As you're making progress breaking up the monolith and implementing those pieces of functionality as services or functions, you update the gateway to redirect requests to the new functionality, instead as shown in Figure 6-2.

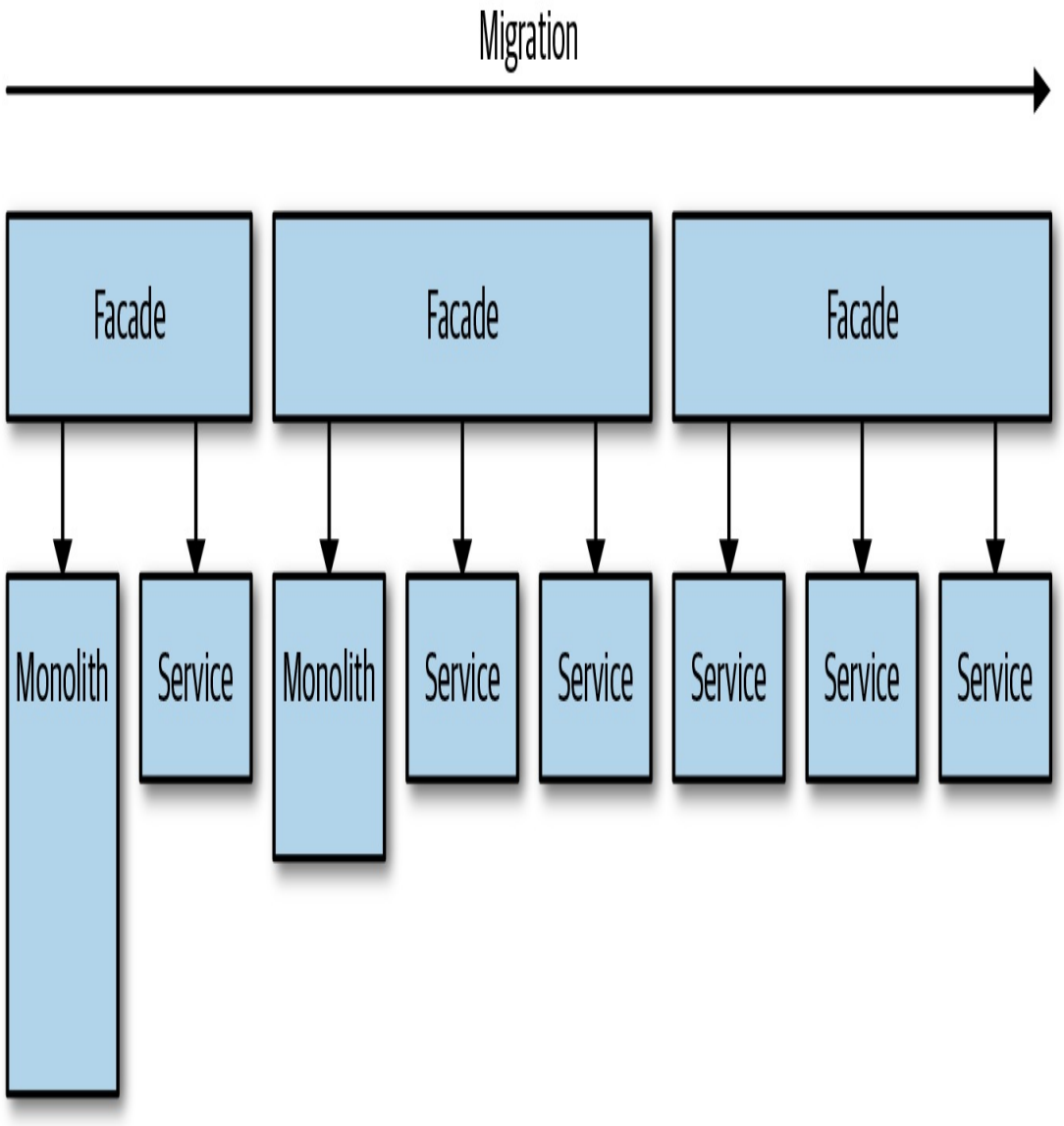

Figure 6-2. Migrating from monolith using the Strangler pattern

Note that the Strangler pattern might not be suitable for the instance in which you can’t intercept the requests going to the backing monolith. The pattern also might not make sense if you have a smaller system, for which it’s easier and faster to replace the entire system, instead of gradually moving it.

The Anticorruption Layer and Strangler patterns have been proven many times as good approaches to move a monolithic legacy application to a

cloud native application because both promote a gradual approach.

Come Up with a Data Migration Strategy

In a monolith, you are usually working with a centrally shared datastore where data is read from and written to by multiple places and services. To truly move to the cloud native architecture, you need to decouple data as well. Your data migration strategy might consist of multiple phases, especially if you can’t migrate everything at the same time. However, in most cases, you will need to do an incremental migration while keeping the entire system running. A gradual migration will probably involve writing data twice (to the new and old datastore) for a while. After you have data in both places and synchronized, you will need to modify where the data is being read from and then read everything from the new store. Finally, you should be able to stop writing data to the old store completely.

Rewrite Any Boilerplate Code

Monoliths will usually have large amounts of code that deals with the configuration, data caching, datastore access, and so on and is probably using older libraries and frameworks. When moving capabilities to a new service, you should rewrite this code. The best option is to throw away the old code and rewrite it from scratch instead of modifying the existing code and molding it so it fits the new service.

Reconsider Frameworks, Languages, Data Structures, and Datastores

Moving to microservices gives you an option to rethink the existing implementation. Are there new frameworks or languages that you could use to rewrite the current code that provide better features and

functionalities for your scenarios? If it makes sense to rewrite the code, do it! Also, reconsider any data structures in the current code. Would they still make sense when moved to a service? You should also evaluate whether you want to use different datastores. Chapter 4 outlines what datastores are best suited for certain data structures and query patterns.

Retire Code

After you've created a new service and all the traffic is redirected to that service, you need to retire and remove the old code that resides in the monolith. Using this approach, you are shrinking the monolith and expanding your services.

Ensuring Resiliency

Resiliency is the ability of a system to recover from failures and continue to function and serve requests. Resiliency is not about avoiding failures; instead, it is all about responding to failures in such a manner that avoids significant downtime or data loss.

Handle Transient Failures with Retries

Requests can fail due to multiple reasons such as network latency, dropped connections, or timeouts if downstream services are busy. You can avoid most of these failures if you retry the request. Retrying can also improve the stability of your application. However, before blindly retrying all requests, you need to implement a bit of logic that determines whether the request should be retried. If the failure is not transient or there is a likelihood that a retry won't be successful, it is better for the component to cancel the request and respond with an appropriate error message. For example, retrying a failed login because of an incorrect password is futile

and retries won’t help. If failure is due to a rare network issue, you can retry the request right away given that the same issue probably won’t persist. Finally, if the failure happens because the downstream service is busy or you are being rate limited, for example, you should retry after a delay. Here are some common strategies for delaying between retry operations:

Constant

: Wait for the same time between each attempt.

Linear

: Incrementally increase the time between each retry. For example, you can start with one second, then three seconds, five seconds, and so on.

Exponential back-off

: Exponentially increase time between each retry. For example, start with 3 seconds, 12 seconds, 30 seconds, and so on.

Depending on what type of failure you are dealing with, you can also immediately retry the operation once and then use one of the delay strategies mentioned in the preceding list. You can handle retries in the component’s source code by using the retry and transient failure logic provided by many of the service SDKs, or at the infrastructure layer if you are using a service mesh, such as Istio.

Use a Finite Number of Retries

Regardless of which retry strategy you’re using, always make sure to use a finite number of retries. Having an infinite number of retries will cause an unnecessary strain on the system.

Use Circuit Breakers for Nontransient Failures

The purpose of a circuit breaker is to prevent components from doing operations that will likely fail and are not transient. Circuit breakers monitor the number of faults, and based on that information decide whether the request should continue or an error should be returned without even invoking the downstream service. If a circuit breaker trips, the number of failures has exceeded a predefined value, and the circuit breaker will automatically return errors for a preset time. After the preset time elapses, it will reset the failure count and allow requests to go through to the downstream service again. A well-known library that implements the circuit breaker pattern is Hystrix from Netflix. If you are using a service mesh like Istio or Envoy proxies, you can take advantage of the circuit breaker implementation in those solutions.

Graceful Degradation

Services should degrade gracefully, so even if they fail, they still provide an acceptable user experience if it makes sense. For example, if you can’t retrieve the data, you could display a cached version of the data, and as soon as the data source recovers, you show the latest data.

Use a Bulkhead Pattern

The *Bulkhead* pattern refers to isolating different parts of your system into groups in such a way that if one fails, the others will continue running unaffected. Grouping your services this way allows you isolate failures and continue serving requests even when there’s a failure.

Implement Health Checks and Readiness Checks

Implement a health check and a readiness check for every service you deploy. The platform can use these to determine whether the service is

healthy and performing correctly as well as when the service is ready to start accepting requests. In Kubernetes, health checks are called *probes*. The liveness probe is used to determine when a container should be restarted, whereas the readiness probe determines whether a pod should start receiving traffic.

The initial delay defines the number of seconds after the container has started before liveness or readiness probes are active, whereas the period defines how often the probe is performed. There are also additional settings such as success/failure threshold and timeouts that you can use to fine-tune the probes.

Define CPU and Memory Limits for Your Containers

You should define CPU and memory limits to isolate resources and prevent certain services instances from consuming too many resources. In Kubernetes, you can achieve this by defining the memory and CPU limits within the pod definition.

Implement Rate Limiting and Throttling

You use rate limiting and throttling to limit the number of incoming or outgoing requests for a service. Implementing those can help you to keep your service responsive even in the case of a sudden spike in requests. Throttling, on the other hand, is often used for outgoing requests. Think about using it when you want to control the number of requests sent to an external service to minimize the costs or to make sure that your service does not look like the origin of a Denial-of-Service attack.

Ensuring Security

Security in the cloud native world is based on the shared responsibility model. The cloud providers are not solely responsible for the security of their customers' solutions; instead, they share that responsibility with the customers. From an application perspective you should consider adopting the defense-in-depth concept, which is discussed in Chapter 3. The best practices listed in this section will help you to ensure security.

Treat Security Requirements the Same as Any Other Requirements

Having fully automated processes is in spirit of the cloud native development. To achieve this, all security requirements must be treated as any other requirement and be pushed through your development pipeline.

Incorporate Security in Your Designs

As you're planning and designing your cloud native solutions, you need to think about security and incorporate the security features in your design. As part of your design, you also should call out any additional security concerns that need to be addressed during component development.

Grant Least-Privileged Access

If your services or functions need access to any resources, they should be granted specific permissions that have the least amount of access set to them. For example, if your service is reading only from the database, it does not need to use an account that has write permissions.

Use Separate Accounts/Subscriptions/Tenants

Depending on the terminology of your cloud provider, your cloud native system should use separate accounts, subscriptions, and/or tenants. At the

very least, you will need a separate account for every environment you will be using; that way, you can ensure proper isolation between environments.

Securely Store All Secrets

Any secrets within your system, used either by your components or Continuous Integration/Continuous Development (CI/CD) pipeline, need to be encrypted and securely stored. It might sound like a no-brainer, but never store any secrets in plain text: always encrypt them. It's always best to use existing and proven secret management systems that take care of these things for you. The simplest option is to use Kubernetes Secrets to store the secrets used by services within the cluster. Secrets are stored in etcd, a distributed key/value store. However, managed and centralized solutions have multiple advantages over Kubernetes secrets: everything is stored in a centralized location, you can define access control policies, secrets are encrypted, auditing support is provided, and more. Some examples of managed solutions are Microsoft Azure Key Vault, Amazon Secrets Manager, and HashiCorp Vault.

Obfuscate Data

Any data your component uses needs to be properly obfuscated. For example, you never want to log any data classified as Personally Identifiable Information (PII) in plain text; if you need to log or store it, ensure that it's either obfuscated (if logging it) or encrypted (if storing it).

Encrypt Data in Transit

Encrypting data in transit protects your data if communications are intercepted while the data moves between components. To achieve this

protection, you need to encrypt the data before transmitting it, authenticate the endpoints, and finally decrypt and verify the data after it reaches the endpoint. Transport Layer Security (TLS) is used to encrypt data in transit for transport security. If you are using a service mesh, TLS might already be implemented between the proxies in the mesh.

Use Federated Identity Management

Using an existing federated identity management service (Auth0, for example) to handle how users sign up, sign in, and sign out allows you to redirect users to a third-party page for authentication. Your component should delegate authentication and authorization whenever possible.

Use Role-Based Access Control

Role-Based Access Control (RBAC) has been around for a long time. RBAC is a control access mechanism around roles and privileges, and as you have learned, it can be a great asset to your defense-in-depth strategy because it allows you to provide fine-grained access to users to only the resources they need. Kubernetes RBAC, for example, controls permissions to the Kubernetes API. Using RBAC, you can allow or deny specific users from creating deployments or listing pods, and more. It’s a good practice to scope Kubernetes RBAC permissions by namespaces rather than cluster roles.

Isolate Kubernetes Pods

Any pods running in a Kubernetes cluster are not isolated and can accept requests from any source. Defining a network policy on pods allows you to isolate pods and make them reject any connections that are not allowed by the policy. For example, if a component in your system is

compromised, a network policy will prevent the malicious actor from communicating with services with which you don’t want them to communicate. Using a NetworkPolicy resource in Kubernetes, you can define a pod selector and detailed ingress and egress policies.

Working with Data

Most modern applications have some need to store and work with data. A growing number of data storage and analytics services are available as cloud provider–managed services. Cloud native applications are designed to take full advantage of cloud provider–managed data systems and are designed to evolve to take advantage of a growing number of features. When working with data in the cloud, many of the standard data best practices still apply: have a disaster recovery plan, keep business logic out of the database, avoid overfetching or excessively chatty I/O, use data access implementations that prevent SQL injections attacks, and so on.

Use Managed Databases and Analytics Services

Whenever possible use a managed database. Provisioning a database on virtual machines (VMs) or in a Kubernetes cluster can often be a quick and easy task. Production databases that require backups and replicas can quickly increase the time and burden of operating data storage systems. By offloading the operational burden of deploying and managing a database, teams are able to focus more on development.

In some cases, a data storage technology might not be available as a managed service or it might be necessary to have access to some configurations that are not available in a managed version of the system.

Use a Datastore That Best Fits Data Requirements

When designing on-premises applications, architects would often try to avoid using multiple databases. Each database technology used would require database administrators with the skillset to deploy and manage the database, significantly increasing the operational costs of the application. The reduced operational costs of cloud-managed databases make it possible to use multiple different types of datastores to put data in a system best suited for the data type, read, and write requirements. Cloud native applications take full advantage of this, using multiple data storage technologies.

Keep Data in Multiple Regions or Zones

Store production data for applications across multiple regions or zones. How the data is stored across the zones or regions will depend on the application's availability requirements; for example, the data might be backups or a replicated database. If a cloud provider experiences a failure of a zone or region, the data can be available to be used for recovery or failover.

Use Data Partitioning and Replication for Scale

Cloud native applications are designed to scale out as opposed to scale up. Scaling a database up is achieved by increasing the resources available to a database instance; for example, adding more cores or memory. This ultimately encounters a hard limit and can be costly. Scaling databases out is achieved through distributing the data across multiple instances of a database. The database is partitioned, or broken up, and stored in multiple databases.

Avoid Overfetching and Chatty I/O

Overfetching is when an application requests data from a database but needs only a fraction of the data for the operation. For example, an application might display a list of orders with a simple summary but request the entire order and order details without needing it. A chatty application, on the other hand, makes a lot of small calls to complete an operation when a single request can be made to the database.

Don’t Put Business Logic in the Database

Too many application scaling issues are the result of putting too much logic in the database. Databases made it easy to put business logic inside the database by supporting standard development languages, and it became convenient to perform these tasks in the database. This often introduces scaling issues because a database is commonly an expensive shared resource.

Test with Production-like Data

Create automation to anonymize production data that can be updated with new rules as the data changes. Applications should be tested with production-like data. Data is sometimes pulled from production systems, scrubbed, and loaded into test systems to provide production-like data. You should automate this process so that it is easy to update as the data changes.

Handle Transient Failures

As mentioned in the resiliency section of this chapter, failures will happen. Expect failures when making calls to a database and be prepared to handle them. Many of the database client libraries support transient fault handling

already. It’s important to understand whether they do and how it’s supported.

Performance and Scalability

Performance indicates how well a system can execute an operation within a certain time frame, whereas scalability refers to how a system can handle load increase without impact on the performance. Predicting periods of increased activity to a system can be tough, so the components need to be able to scale out as needed to meet the increased demand and then scale down, after the demand decreases. The subsections that follow present some best practices to help you achieve optimal performance and scalability.

Design Stateless Services That Scale Out

Services should be designed to scale out. Scaling out is an approach to increasing the scale of a service by adding more instances of a service. Scaling up is an approach to scaling a service by adding more resources like memory or cores, but this method generally has a hard limit. By designing a service to scale out and back in, you can scale the service to handle variations in the load without impacting the availability of the service.

Stateful applications are inherently difficult to scale and should be avoided. If stateful services are necessary, it’s generally best to separate the functionality from the application and use a partitioning strategy and managed services if they are available.

Use Platform Autoscaling Features

When possible, use any autoscaling features that are built into the platform before implementing your own. Kubernetes offers Horizontal Pod Autoscaler (HPA). HPA scales the pods based on the CPU, memory, or custom metrics. You specify the metric (e.g., 85% of CPU or 16 GB of memory) and the minimum and maximum number of pod replicas. After the target metric is reached, Kubernetes automatically scales the pods. Similarly, cluster autoscaling scales the number of cluster nodes if pods can't be scheduled. Cluster autoscaling uses the requested resources in the pod specification to determine whether nodes should be added.

Use Caching

Caching is a technique that can help improve the performance of your component by temporarily storing frequently used data in storage that's close to the component. This improves the response time because the component does not need to go to the original source. The most basic type of cache is an in-memory store that is being used by a single process. If you have multiple instances of your component, each instance will have its own independent copy of the in-memory cache. This can cause consistency problems if data is not static because the different instances will have different versions of cached data. To solve this problem, you can use shared caching, which ensures that different component instances use the same cached data. In this case, cache is stored separately, usually in front of the database.

Use Partitioning to Scale Beyond Service Limits

Cloud services will often have some defined scale limits. It's important to understand the scalability limits of each of the services used and how much they can be scaled up. If a single service is unable to scale to meet the application's requirements, create multiple service instances and

partition work across the instances. For example, if a managed gateway was capable of handling 80% of the application's intended load, create another gateway and split the services across the gateway.

Functions

Much of the software development life cycle (SDLC) and general server architecture best practices are the same for serverless architectures. Given serverless is a different operating model, there are, however, some best practices specific to functions.

Write Single-Purpose Functions

Follow the single-responsibility principle and only write functions that have a single responsibility. This will make your functions easier to reason about, test, and, when the time comes, debug.

Don't Chain Functions

In general, functions should push messages/data to a queue or a datastore to trigger any other functions if needed. Having one or more functions call other functions is often considered an antipattern that additionally increases your cost and makes the debugging more difficult. If your application requires the daisy-chaining of functions, you should consider using function offerings such as Azure Durable Functions or AWS Step Functions.

Keep Functions Light and Simple

Each function should do just one thing and rely on only a minimal number of external libraries. Any extra and unnecessary code in the function

makes the function bigger in size, and that affects the start time.

Make Functions Stateless

Don’t save any data in your functions because new function instances usually run in their own isolated environment and don’t share anything with other functions or invocations of the same function.

Separate Function Entry Point from the Function Logic

Functions will have an entry point invoked by the function framework. Framework-specific context is generally passed to the function entry point, along with invocation context. For example, if the function is invoked through an HTTP request like an API gateway, the context will contain HTTP-specific details. The entry-point method should separate these entry-point details from the rest of the code. This will improve manageability, testability, and portability of the functions.

Avoid Long-Running Functions

Most Function as a Service (FaaS) offerings have an upper limit for execution time per function. As a result, long-running functions can cause issues such as increased load times and timeouts. Whenever possible, refactor large functions into smaller ones that work together.

Use Queues for Cross-Function Communication

Instead of passing information among one another, functions should use a queue to which to post the messages. Other functions can be triggered and executed based off the events that happen on that queue (item added, removed, updated, etc.).

Operations

A DevOps practice provides the foundation necessary for organizations to make the best use of cloud technologies. Cloud native applications utilize DevOps principles and best practices that are detailed in Chapter 5.

Deployments and Releases Are Separate Activities

It is important to make a distinction between deployment and release. Deployment is the act of taking the built component and placing it within an environment—the component is fully configured and ready to go; however, there is no traffic being sent to it. As part of the component release, we begin to allow traffic to the deployed component. This separation allows you to do gradual releases, A/B testing, and canary deployments in a controlled manner.

Keep Deployments Small

Each component deployment should be a small event that can be performed by a single team in a short time. There is no general rule about how small a deployment should be and how much time it should take to deploy a component, because this is highly dependent on the component, your process, and the change to the component. A good approach is to be able to roll out a critical fix within a day.

CI/CD Definition Lives with the Component

You need to store and version any CI/CD configuration and dependencies alongside the component. Each push to the component's branch triggers the pipeline and executes jobs defined in the CI/CD configuration. To control component deployments to different environments (development, staging, production), you can use the Git branch names and configure your

pipeline to deploy the master branch only to a production environment, for example.

Consistent Application Deployment

With a consistently reliable and repeatable deployment process, you can minimize errors. Automate as many processes as possible and ensure that you have a rollback plan defined in case deployment fails.

Use Zero-Downtime Releases

To maximize the availability of your system during releases, consider using zero-downtime releases such as blue/green or canary. Using one of these approaches also allows you to quickly roll back the update in case of failures.

Don’t Modify Deployed Infrastructure

Infrastructure should be immutable. Modifying deployed infrastructure can quickly get out of hand, and keeping track of what changed can be complicated. If you need to update the infrastructure, redeploy it instead.

Use Containerized Build

To avoid configuring build environments, package your build process into Docker containers. Consider using multiple images and containers for builds instead of creating a single, monolithic build image.

Describe Infrastructure Using Code

Infrastructure should be described using either cloud provider’s declarative templates or a programming language or scripts that provision the infrastructure.

Use Namespaces to Organize Services in Kubernetes

Every resource in a Kubernetes cluster belongs to a namespace. By default, newly created resources go into a namespace called *default*. For better organization of services, it is a good practice to use descriptive names and group services into bounded contexts.

Isolate the Environments

Use a dedicated production cluster and physically separate the production cluster for your development, staging, or testing environments.

Separate Function Source Code

Each function must be independently versioned and have its own dependencies. If that’s not the case, you will end up with a monolith and a tightly coupled codebase.

Correlate Deployments with Commits

Pick a branching strategy that allows you to correlate the deployments to specific commits in your branch and that also allows you to identify which version of the source code is deployed.

Logging, Monitoring, and Alerting

Application and infrastructure logging can provide much more value than just root-cause analysis. A proper logging solution will provide valuable insights into applications and systems, and it’s often necessary for monitoring the health of an application and alerting operations of important events. As cloud applications become more distributed, logging

and instrumentation become increasingly challenging and important.

Use a Unified Logging System

Use a unified logging system capable of capturing log messages across all services and levels of a system and store them in a centralized store. Whether you move all logs to a centralized store for analysis and search, or you leave them on the machine with the necessary tools in place to run a distributed query, it’s important that an engineer can find and analyze logs without having to go from one system to the next.

Use Correlation IDs

Include a unique correlation ID (CID) that is passed through all services. If one of the services fails, the correlation ID is used to trace the request through the system and pinpoint where the failure occurred.

Include Context with Log Entries

Each log entry should contain additional context that can help when you are investigating issues. For example, include all exception handling, retry attempts, service name or ID, image version, binary version, and so on.

Common and Structured Logging Format

Decide on a common and structured logging format that all components will use. This will allow you to quickly search and parse the logs later on. Also, make sure you are using the same time zone information in all your components. In general, it is best to adhere to a common time format such as Coordinated Universal Time (UTC).

Tag Your Metrics Appropriately

In addition to using clear and unique metric names, make sure that you are storing any additional information, such as component name, environment, function name, region, and so forth, in the metric tags. With tags in place, you can create queries, dashboards, and reports using multiple dimensions (e.g., average latency across a specific region or across regions for a specific function).

Avoid Alert Fatigue

The sheer number of metrics makes it difficult to determine how to set up the alerting and what to alert on. If you are firing off too many alerts, eventually people will stop paying attention to them and no longer take them seriously. Also, investigating a bunch of alerts can become overwhelming and it could be the only thing your team is doing. It is important to classify alerts by severity: low, moderate, and high. The purpose of low-severity alerts is to potentially use them later, when doing root-cause analysis of a high-severity alert. You can use them to uncover certain patterns, but they do not require any immediate action when fired. A moderate-severity alert should either create a notification or open a ticket. These are the alerts you want to look at, but are not high priority and don’t need immediate action. They could represent a temporary condition (increase demand, for example) that eventually goes away. They also give you an early warning of a possible high-severity alert. Finally, high-severity alerts are the ones that will wake people up in the middle of the night and require immediate action. Recently, machine learning–based approaches to automatically triage issues and raise alerts are gaining in popularity, and the term AIOps has even been introduced.

Define and Alert on Key Performance Indicators

Cloud native systems will have a plethora of signals that are being emitted

and monitored. You need to filter down those signals and determine which ones are the most important and valuable. These Key Performance Indicators (KPIs) give you insight into the health of your system. For example, one KPI is latency, which measures the time it takes to service a request. If you begin seeing latency increase or deviate from an acceptable range, it is probably time to issue an alert and have someone take a look at it. In addition to KPIs, you can use other signals and metrics to determine why something is failing.

Continuous Testing in Production

Using continuous testing you can generate requests that are sent throughout the system and simulate real users. You can utilize this traffic to get test coverage for the components, discover potential issues, and test your monitoring and alerting. Following are some common continuous testing practices:

- Blue/green deployments
- Canary testing
- A/B testing

These practices are discussed in Chapter 5.

Start with Basic Metrics

Ensure that you are always collecting traffic (how much demand is placed on the component), latency (the time it takes to service a request), and errors (rate of requests that fail) for each component in your system.

Service Communication

Service communication is an important part of cloud native applications. Whether it's a client communicating with a backend, a service communicating with a database, or the individual services in a distributed architecture communicating with one another, these interactions are an important part of cloud native applications. Many different forms of communication are used depending on the requirements. The following subsections offer some best practices for service communication.

Design for Backward and Forward Compatibility

With backward compatibility, you ensure that new functionality added to a service or component does not break any existing service. For example, in Figure 6-3, Service A v1.0 works with Service B v1.0. Backward compatibility means that the release of Service B v1.1 will not break the functionality of Service A.

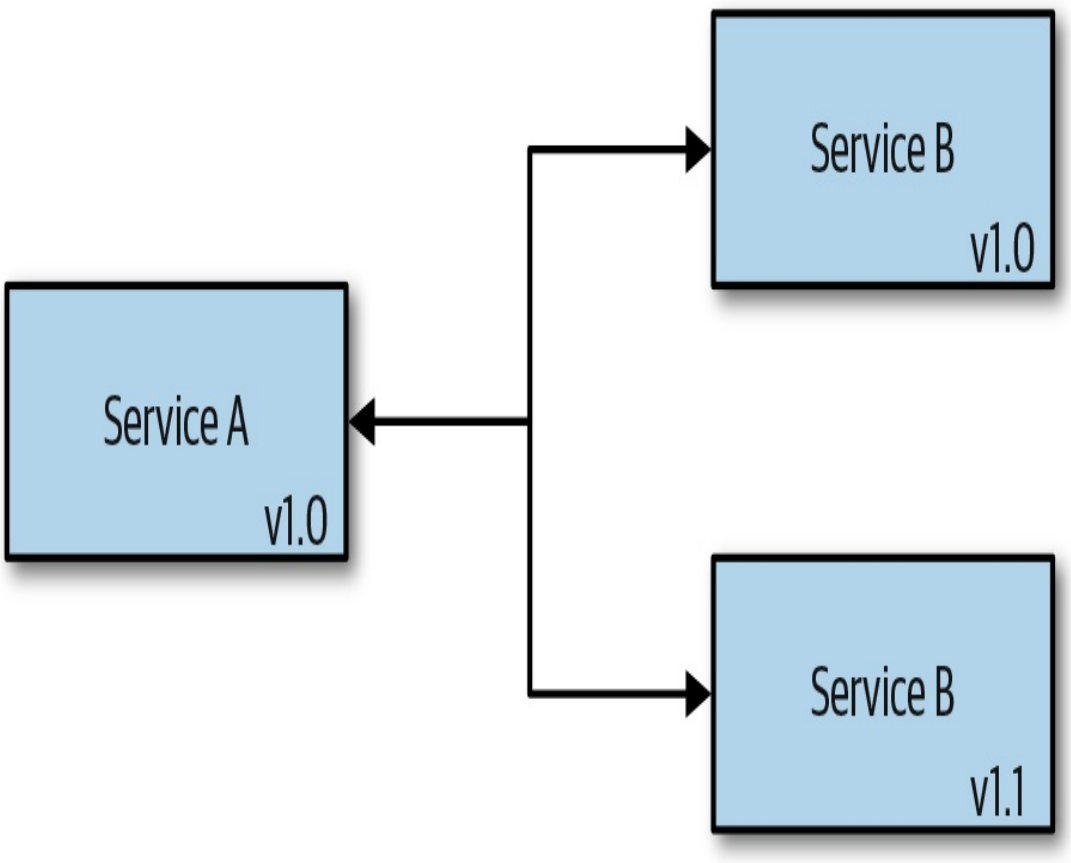

Figure 6-3. Backward compatibility

To ensure backward compatibility, any new fields added to the API should be optional or have sensible defaults. Any existing fields should never be renamed, because that will break the backward compatibility.

> **NOTE**
>
> *Parallel change*, also known as the *Expand and Contract* pattern, can be used to safely introduce backward-incompatible changes. As an example, say a service owner would like to change a property or resource on an interface. The service owner will expand the interface with a new property or resource, and then after all consumers have had a chance to move the service interface, the previous property is removed.

If your system or components need to ensure rollback functionality, you

will need to think about the forward compatibility as you're making changes to your service. Forward compatibility means that your components are compatible with future versions. Your service should be able to accept "future" data and messaging formats and handle them appropriately. A good example of forward compatibility is HTML: when it encounters an unknown tag or attribute, it's not going to fail; it will just skip it.

Define Service Contracts That Do Not Leak Internal Details

A service that exposes an API should define contracts and test against the contracts when releasing updates. For example, a REST-based service would generally define a contract in the OpenAPI format or as documentation, and consumers of the service would build to this contract. Updates to the service can be pushed, and as long as it doesn't introduce any breaking changes to the API contract, these releases would not affect the consumer. Leaking internal implementations of a service can make it difficult to make changes and introduces coupling. Don't assume a consumer is not using some piece of data exposed through the API.

> **NOTE**
>
> Services that publish messages to a queue or a stream should also define a contract in the same way. The service publishing the events will generally own the contract.

Prefer Asynchronous Communication

Use asynchronous communication whenever possible. It works well with distributed systems and decouples the execution of two or more services.

A message bus or a stream is often used when implementing this approach, but you could use direct calls through something like gRPC as well. Both use a message bus as a channel.

Use Efficient Serialization Techniques

Distributed applications like those built using a microservices architecture rely more heavily on communications and messaging between services. The data serialization and deserialization can add a lot of overhead in service communication.

> **NOTE**
>
> In one case, serialization and deserialization were found to account for nearly 40% of the CPU utilization across all the services. Replacing the standard JSON serialization library with a custom one reduced this overhead to roughly 15% of overall CPU utilization.

Use efficient serialization formats like protocol buffers, commonly used in gRPC. Understand the trade-offs with the different serialization formats, because tooling and consumer requirements might not make this a feasible option. You can also use other techniques to reduce the need for serialization in some services by placing some of the data into headers. For example, if a service receives a request and operates on only a handful of fields in a large message payload before passing it to a downstream service, by putting these fields into headers the service does not need to deserialize or reserialize the payload. The service reads and writes headers and then simply passes the entire payload through to the downstream services.

Use Queues or Streams to Handle Heavy Loads and Traffic Spikes

A queue or a stream between components acts as a buffer and stores the message until it is retrieved. Using a queue allows the components to process the messages at their own pace, regardless of the incoming volume or load. Consequently, this helps maximize the availability and scalability of your services.

Batch Requests for Efficiency

Queues can be used for batching multiple requests and performing an action only once. For example, it is more efficient to write 1,000 batched entries into the database instead of one entry at a time 1,000 times.

Split Up Large Messages

Sending, receiving, and manipulating large messages requires more resources and can slow down your entire system. The *Claim-Check* pattern talks about splitting a large message into two parts. You store the entire message in an external service (database, for example) and send only the reference to the message. Any interested message receivers can use the reference to obtain the full message from the database.

Containers

It’s possible to run most applications in a Docker container without very much effort. However, there are some potential pitfalls when running containers in production and streamlining the build, deployment, and monitoring. A number of best practices have been identified to help avoid the pitfalls and improve the results.

Store Images in a Trusted Registry

Any images running on the platform should come from the trusted container image registry. Kubernetes exposes a webhook (validating admission) that can be used to ensure pods can use images only from a trusted registry. If you're using Google Cloud, you can take advantage of the binary authorization security measure that ensures only trusted images are deployed on your cluster.

Utilize the Docker Build Cache

Using the build cache when building Docker images can speed up the build process. All images are built up from layers, and each line in the Dockerfile contributes a layer to the final image. During the build, Docker will try to reuse a layer from a previous build instead of building it again. However, it can reuse only the cached layers if all previous build steps used it as well. To get the most out of the Docker build cache, put the commands that change more often (e.g., adding the source code to the image, building the source code) at the end of the Dockerfile. That way, any preceding steps will be reused.

Don't Run Containers in Privileged Mode

Running containers in privileged mode allows access to everything on the host. Use the security policy on the pod to prevent containers from running in privileged mode. If a container does for some reason require privileged mode to make changes to the host environment, consider separating that functionality from the container and into the infrastructure provisioning.

Use Explicit Container Image Tags

Always tag your container images with specific tags that tightly link the container image to the code that is packaged in the image. To tag the images properly, you can either use a Git commit hash that uniquely identifies the version of the code (e.g., `1f7a7a472`) or use a semantic version (e.g., `1.0.1`). The tag `latest` is used as a default value if no tag is provided; however, because it's not tightly linked to the specific version of the code, you should avoid using it. The latest tag should never be used in a production environment because it can cause inconsistent behavior that can be difficult to troubleshoot.

Keep Container Images Small

In addition to taking up less space in a container registry or the host system using the image to run a container, smaller images improve image push and pull performance. This in turn improves the performance when you start containers as part of deploying or scaling a service. The application and its dependencies will have some impact on the size of the image, but you can reduce most of the image size by using lean base images and ensuring that unnecessary files are not included in the image. For example, the alpine 3.9.4 image is only 3 MB, with the Debian stretch image at 45 MB, and the CentOS 7.6.1810 at 75 MB. The distributions generally offer a slim version that removes more from the base image that might not be needed by the application. Generally, there are two things to keep in mind for keeping images lean:

- Start with a lean base image
- Include only the files needed for the operation of the application

You can use the Container Builder pattern to create lean images by separating the images used to build the artifacts from the base image used to run the application. Docker's multistage build is often used to

implement this. You can create Docker build files that can start from different images used for executing the commands to build and test artifacts, and then define another base image as part of creating the image to run the application.

> **TIP**
>
> Using a *.dockerignore* file can improve build speed by excluding files that are not needed in the Docker build.

Run One Application per Container

Always run a single application within a container. Containers were designed to run a single application, with the container having the same life cycle as the application running in the container. Running multiple applications within the same container makes it difficult to manage, and you might end up with a container in which one of the processes has crashed or is unresponsive.

Use Verified Images from Trusted Repositories

There's a large and growing number of publicly available images that are helpful when working with containers. Docker repository tags are mutable, so it's important to understand that the images can change. When using images in an external repository it's best to copy or re-create them from the external repository into one managed by the organization. The organization's repository is usually closer to the CI services, and this approach removes another service dependency that could impact build.

Use Vulnerability Scanning Tools on Images

You need to be aware of any vulnerabilities that affect your images because this can compromise the security of your system. If a vulnerability is discovered, you need to rebuild the image with the patches and fixes included and then redeploy it. Some cloud providers offer vulnerability scanning with their image registry solutions, so make sure you are taking advantage of those features.

> **TIP**
>
> Scan an image as often as possible because new cybersecurity vulnerabilities and exposures (CVE) are released daily.

Don’t Store Data in Containers

Containers are ephemeral—they can be stopped, destroyed, or replaced without any loss of data. If the service running in a container needs to store data, use a volume mount to save the data. The contents in a volume exist outside the life cycle of a container and a volume does not increase the size of a container. If the container requires temporary nonpersistent writes, use a tmpfs mount, which will improve performance by avoiding writes to a container’s writable layer.

Never Store Secrets or Configuration Inside an Image

Hardcoding any type of secrets within an image is something you want to avoid. If your container requires any secrets, define them within environment variables or as files, mounted to the container through a volume.

Summary

We could easily fill an entire book covering best practices for cloud native applications given the number of technologies involved. However, there are certain areas that have been coming up repeatedly in customer conversations, and this chapter has covered a collection of best practices, tips, and proven patterns for cloud native applications for those areas. You should have a better understanding of the factors you may want to consider.

Chapter 7. Portability

Portability is sometimes a concern when building cloud native applications. The application might have a requirement to be deployed across multiple cloud providers or even on-premises. These requirements are generally driven by stakeholders, whether they are customers using the application or the business building the application. It might be the case that the application is deployed by the customer, either on their own hardware or in their own account on the cloud provider of their choice. Regardless of the reasons, the requirement for portability should be treated like any other architecturally significant requirement. It should be driven by the business with careful consideration to the costs and trade-offs.

Why Make Applications Portable?

There are many good reasons to make applications portable. Portability should be a requirement, and the trade-offs and costs associated with the feature should be considered. Following are some of the reasons why software vendors make applications portable:

- Building an application that's deployed into a customer's environment and there's a requirement to offer deployment into the customer's choice of cloud provider or on-premises.
- Building a hybrid application that runs in the cloud and on-premises, where some of the services in the application run in both environments.
- Services need to be near a customer's application in order to minimize latency. These could be services that store or analyze

data, for example. ...

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

About the Authors

Boris Scholl is a lead product architect with the Azure Compute engineering team focusing on the next generation of distributed systems platforms and application models. He has been working on Azure Developer tools and platforms in various product engineering roles since late 2011. Boris re-joined the Azure Compute team in 2018, after having spent the 18 months outside Microsoft leading an engineering team to work on a microservices platform based on Kubernetes and service meshes. His work on distributed systems platforms has resulted in several patents about cloud computing and distributed systems. Boris is a frequent speaker at industry events, a contributor to many blogs, instructor for distributed computing topics, and the lead author of one of the first books about microservices and Docker on Azure, *Microservices with Docker on Azure* (O'Reilly 2016).

Trent Swanson is a software architect focused on cloud and edge technologies. As a Distinguished Fellow of Cloud Technologies at Johnson Controls, he works with a wide range of cloud technologies and a lot of very smart and passionate people to create intelligent buildings. He has helped teams build and operate large and small applications across multiple cloud providers using modern practices and technologies. As a cofounder and consultant with Full Scale 180, he worked with some of Microsoft's largest customers, helping them migrate and build applications in the cloud. He has been involved in architecting, ...

Colophon

The animal on the cover of *Cloud Native* is a purple sandpiper (*Calidris maritima*), a plump shorebird with a large range across arctic and subarctic tundra habitats in North America and Europe. They winter along the rocky coasts of the Atlantic and have the northernmost winter range of any shorebird.

Adults are mostly gray with a slight purplish gloss. They have short, yellow legs and a medium-sized, slightly downcurved bill. On average, they are 9 inches long and weigh 2.5 ounces. Males and females are similar in appearance.

The male purple sandpiper shares responsibility for incubation and then assumes parental care of the hatchlings, which is unusual among monogamous shorebirds. The precocious hatchlings are capable of walking and pecking at the ground for food within a few hours of hatching. Purple sandpipers eat mostly insects, mollusks, spiders, and seeds.

A common behavior of the purple sandpiper and other wading birds is the rodent run, which is a distraction display used to protect the nest from predators. The bird ruffles its feathers, crouches, and runs away from the predator while making a squealing noise that sounds like a mouse. The action resembles the flight response of a small rodent and lures the predator away from the nest.

Many of the animals on O’Reilly covers are endangered; all of them are important to the world.

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