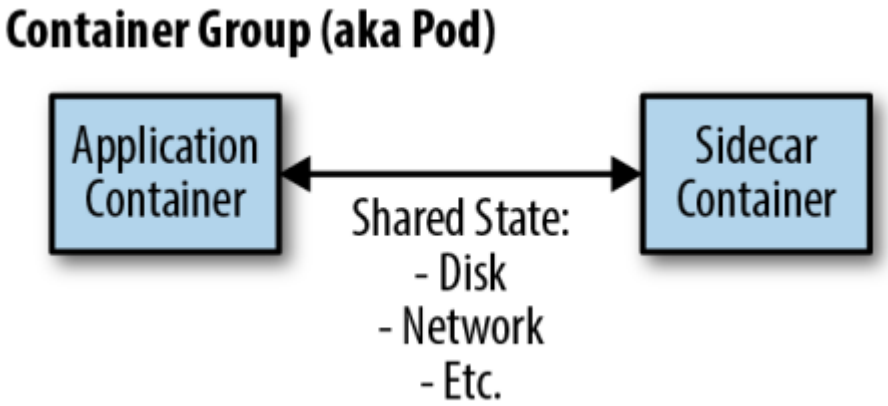
# Microservice

## 1. Distributed system

### 1.1 Single-Node Patterns

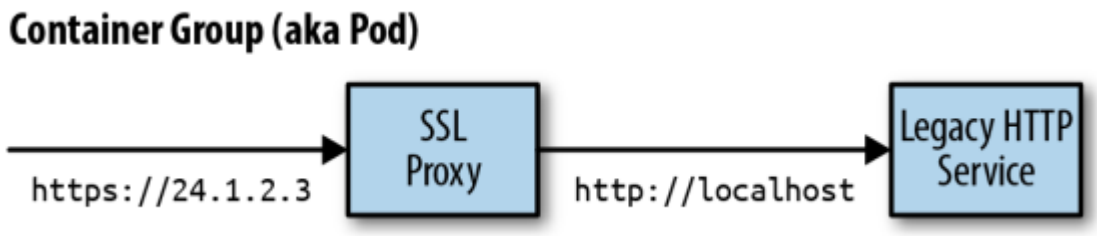
#### a. Side car patterns




The first single-node pattern is the sidecar pattern. The sidecar pattern is a single-node pattern made up of two containers. The first is the *application container*. It contains the core logic for the application. Without this container, the application would not exist. In addition to the application container, there is a *sidecar container*. The role of the sidecar is to augment and improve the application container, often without the application container's knowledge. In its simplest form, a sidecar container can be used to add functionality to a container that might otherwise be difficult to improve.

Example 1 Adding HTTPS to a Legacy Service:

The application of the sidecar pattern to this situation is straightforward. The legacy web service is configured to serve exclusively on localhost (127.0.0.1), which means that only services that share the local network with the server will be able to access the service. Normally, this wouldn't be a practical choice because it would mean that no one could access the web service. However, using the sidecar pattern, in addition to the legacy container, we will add an nginx sidecar container. This nginx container lives in the same network namespace as the legacy web application, so it can access the service that is running on localhost. At the same time, this nginx service can ter-



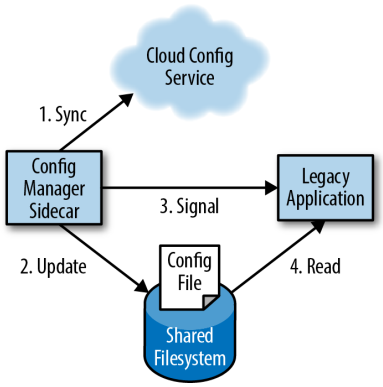

Example 2 Dynamic Configuration with Sidecars

Simply proxying traffic into an existing application is not the only use for a sidecar. Another common example is configuration synchronization. Many applications use a configuration file for parameterizing the application; this may be a raw text file or something more structured like XML, JSON, or YAML. Many pre-existing applications were written to assume that this file was present on the filesystem and read their configuration from there. However, in a cloud-native environment it is often quite useful to use an API for updating configuration. This allows you to do a dynamic push of configuration information via an API instead of manually logging in to every server and updating the configuration file using imperative commands. The desire

for such an API is driven both by ease of use as well as the ability to add automation like rollback, which makes configuring (and reconfiguring) safer and easier.

Similar to the case of HTTPS, new applications can be written with the expectation that configuration is a dynamic property that should be obtained using a cloud API, but adapting and updating an existing application can be significantly more challenging. Fortunately, the sidecar pattern again can be used to provide new functionality that augments a legacy application without changing the existing application. For the sidecar pattern shown in Figure 2-3, there again are two containers: the container that is the serving application and the container that is the configuration manager. The two containers are grouped together into a pod where they share a directory between themselves. This shared directory is where the configuration file is maintained.

When the legacy application starts, it loads its configuration from the filesystem, as expected. When the configuration manager starts, it examines the configuration API and looks for differences between the local filesystem and the configuration stored in the API. If there are differences, the configuration manager downloads the new configuration to the local filesystem and signals to the legacy application that it should reconfigure itself with this new configuration. The actual mechanism for this notification varies by application. Some applications actually watch the configuration file for changes, while others respond to a SIGHUP signal. In extreme cases, the configuration manager may send a SIGKILL signal to abort the legacy application. Once aborted, the container orchestration system will restart the legacy application, at which point it will load its new configuration. As with adding HTTPS to an existing application, this pattern illustrates how the sidecar pattern can help adapt pre-existing applications to more cloud-native scenarios.




## Example PaaS with Sidecars

The sidecar pattern can be used for more than adaptation and monitoring. It can also be used as a means to implement the complete logic for your application in a simplified, modular manner. As an example, imagine building a simple platform as a service (PaaS) built around the `git` workflow. Once you deploy this PaaS, simply pushing new code up to a Git repository results in that code being deployed to the running servers. We'll see how the sidecar pattern makes building this PaaS remarkably straightforward.

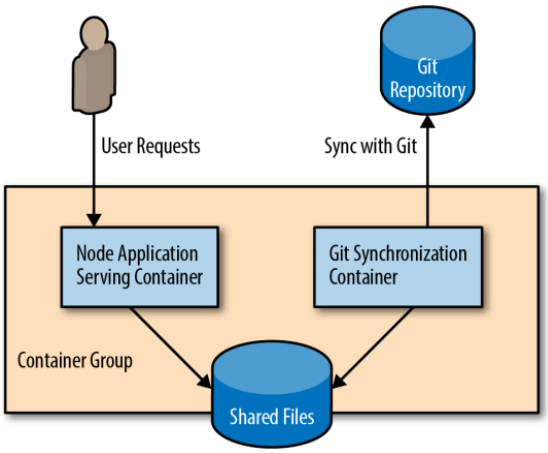
The sidecar container shares a filesystem with the main application container and runs a simple loop that synchronizes the filesystem with an existing Git repository:

```
#!/bin/bash

while true; do
  git pull
  sleep 10
done
```

Obviously this script could be more complex, pulling from a specific branch instead of simply from HEAD. It is left purposefully simple to improve the readability of this example.

The Node.js application and Git synchronization sidecar are coscheduled and deployed together to implement our simple PaaS (Figure 2-4). Once deployed, every time new code is pushed to a Git repository, the code is automatically updated by the sidecar and reloaded by the server.




Designing Sidecars for Modularity and Reusability

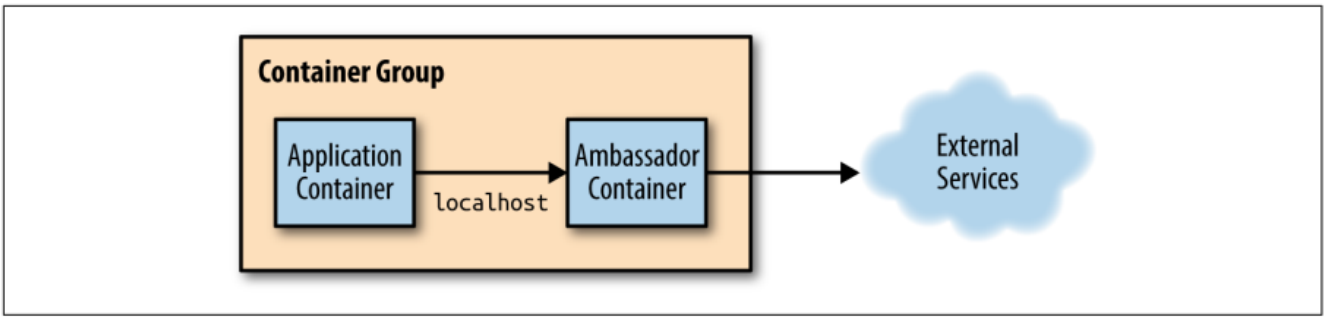
In all of the examples of sidecars that we have detailed throughout this chapter, one of the most important themes is that every one was a modular, reusable artifact. To be successful, the sidecar should be reusable across a wide variety of applications and deployments. By achieving modular reuse, sidecars can dramatically speed up the building of your application.

However, this modularity and reusability, just like achieving modularity in high-quality software development requires focus and discipline. In particular, you need to focus on developing three areas:

- Parameterizing your containers
- Creating the API surface of your container
- Documenting the operation of your container

b. Ambassadors

The previous chapter introduced the sidecar pattern, where one container augments a pre-existing container to add functionality. This chapter introduces the ambassador pattern, where an ambassador container brokers interactions between the application container and the rest of the world. As with other single-node patterns, the two containers are tightly linked in a symbiotic pairing that is scheduled to a single machine. A canonical diagram of this pattern is shown in Figure 3-1.




Example 1 Using an Ambassador for Service Brokering

When trying to render an application portable across multiple environments (e.g., public cloud, physical datacenter, or private cloud), one of the primary challenges is service discovery and configuration. To understand what this means, imagine a front-end that relies on a MySQL database to store its data. In the public cloud, this MySQL service might be provided as software-as-a-service (SaaS), whereas in a private cloud it might be necessary to dynamically spin up a new virtual machine or container running MySQL.

Consequently, building a portable application requires that the application know how to introspect its environment and find the appropriate MySQL service to connect to. This process is called *service discovery*, and the system that performs this discovery and linking is commonly called a *service broker*. As with previous examples, the ambassador pattern enables a system to separate the logic of the application container

from the logic of the service broker ambassador. The application simply always connects to an instance of the service (e.g., MySQL) running on localhost. It is the responsibility of the service broker ambassador to introspect its environment and broker the appropriate connection. This process is shown in Figure 3-3.

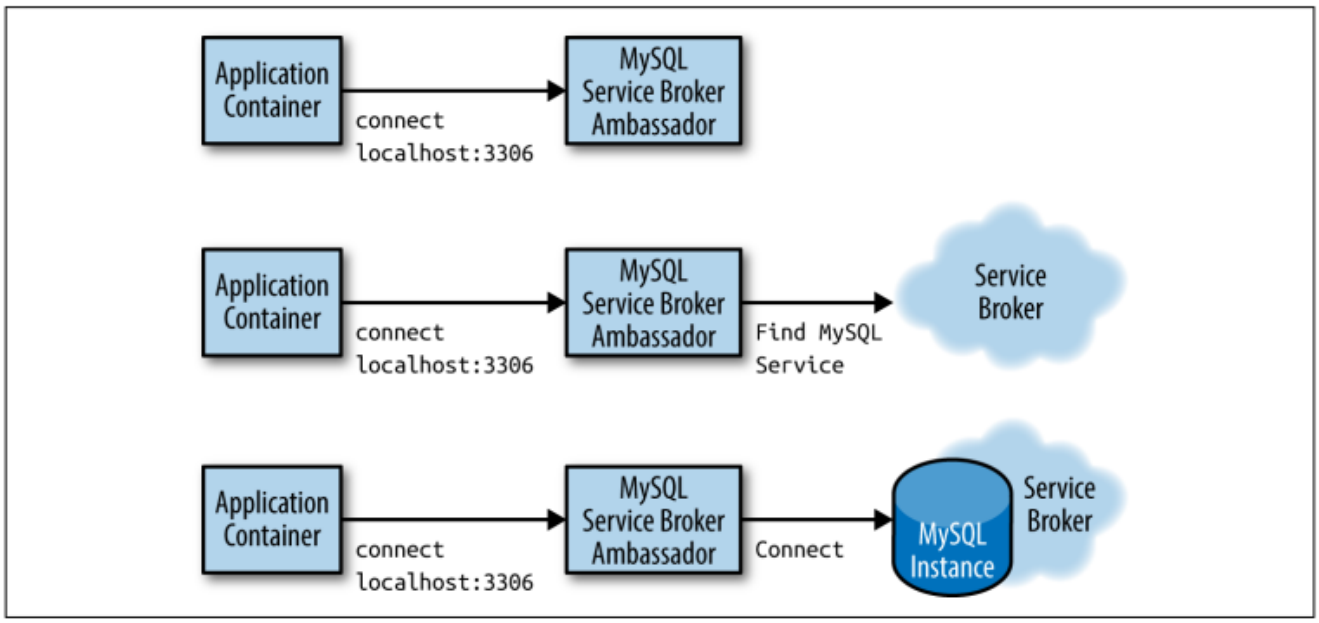


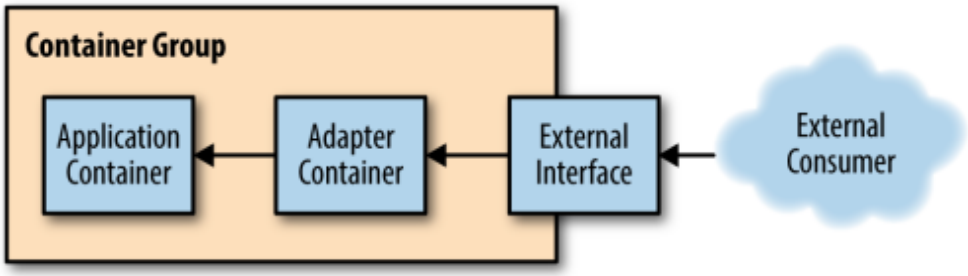

*Figure 3-3. A service broker ambassador creating a MySQL service*

A final example application of the ambassador pattern is to perform experimentation or other forms of request splitting. In many production systems, it is advantageous to be able to perform request splitting, where some fraction of all requests are not serviced by the main production service but rather are redirected to a different implementation of the service. Most often, this is used to perform experiments with new, beta versions of the service to determine if the new version of the software is reliable or comparable in performance to the currently deployed version.

Additionally, request splitting is sometimes used to `tee` or split traffic such that all traffic goes to both the production system as well as a newer, undeployed version. The responses from the production system are returned to the user, while the responses from the `tee-d` service are ignored. Most often, this form of request splitting is used to simulate production load on the new version of the service without risking impact to existing production users.

c. Adapters

In the preceding chapters, we saw how the sidecar pattern can extend and augment existing application containers. We also saw how ambassadors can alter and broker how an application container communicates with the external world. This chapter describes the final single-node pattern: the *adapter* pattern. In the adapter pattern, the *adapter container* is used to modify the interface of the *application container* so that it conforms to some predefined interface that is expected of all applications. For example, an adapter might ensure that an application implements a consistent monitoring interface. Or it might ensure that log files are always written to `stdout` or any number of other conventions.




Example 1 Monitoring

When monitoring your software, you want a single solution that can automatically discover and monitor any application that is deployed into your environment. To make this feasible, every application has to implement the same monitoring interface. There are numerous examples of standardized monitoring interfaces, such as `syslog`, event tracing on Windows (etw), JMX for Java applications, and many, many other protocols and interfaces. However, each of these is unique in both protocol for communication as well as the style of communication (push versus pull).

Example 2 Logging

Much like monitoring, there is a wide variety of heterogeneity in how systems log data to an output stream. Systems might divide their logs into different levels (such as debug, info, warning, and error) with each level going into a different file. Some might simply log to `stdout` and `stderr`. This is especially problematic in the world of containerized applications where there is a general expectation that your containers will log to `stdout`, because that is what is available via commands like `docker logs` or `kubectl logs`.

Adding further complexity, the information logged generally has structured information (e.g., the date/time of the log), but this information varies widely between different logging libraries (e.g., Java's built-in logging versus the `glog` package for Go).

Example 3 Adding a Health Monitor

One last example of applying the adapter pattern is derived from monitoring the health of an application container. Consider the task of monitoring the health of an off-the-shelf database container. In this case, the container for the database is supplied by the database project, and we would rather not modify that container simply to add health checks. Of course, a container orchestrator will allow us to add simple health checks to ensure that the process is running and that it is listening on a particular port, but what if we want to add richer health checks that actually run queries against the database?

The answer to these problems should be easy to guess by now: we can use an adapter container. The database runs in the application container and shares a network interface with the adapter container. The adapter container is a simple container that only contains the shell script for determining the health of the database. This script can then be set up as the health check for the database container and can perform whatever rich health checks our application requires. If these checks ever fail, the database will be automatically restarted.

## 1.2 Serving Patterns

### a. Microservices

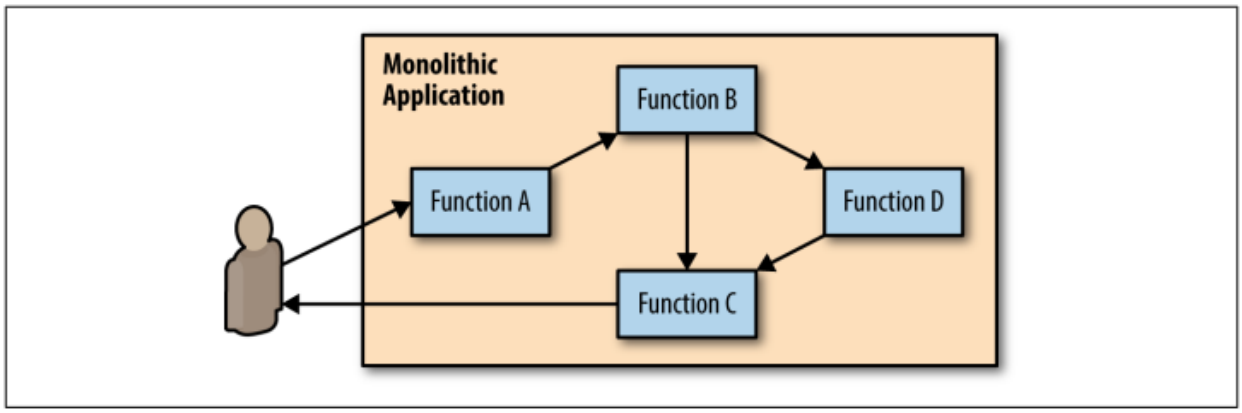
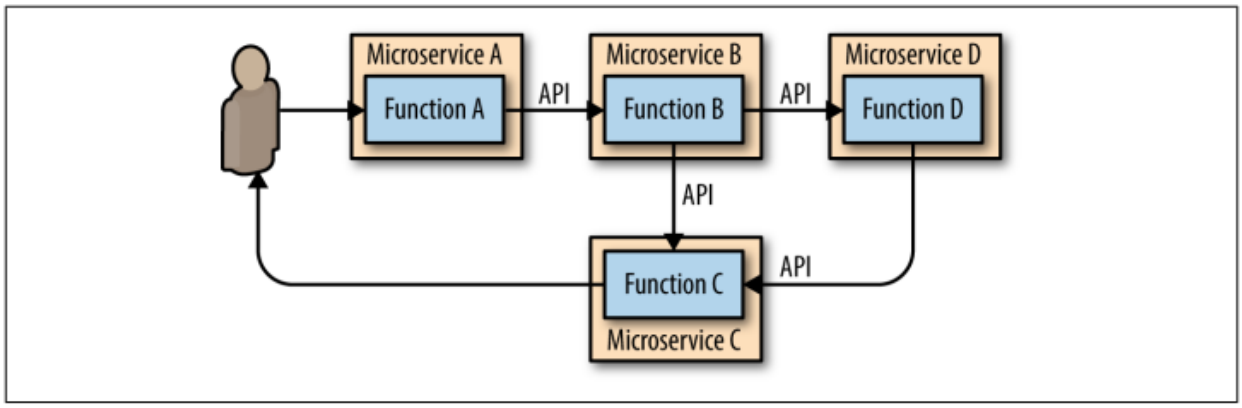


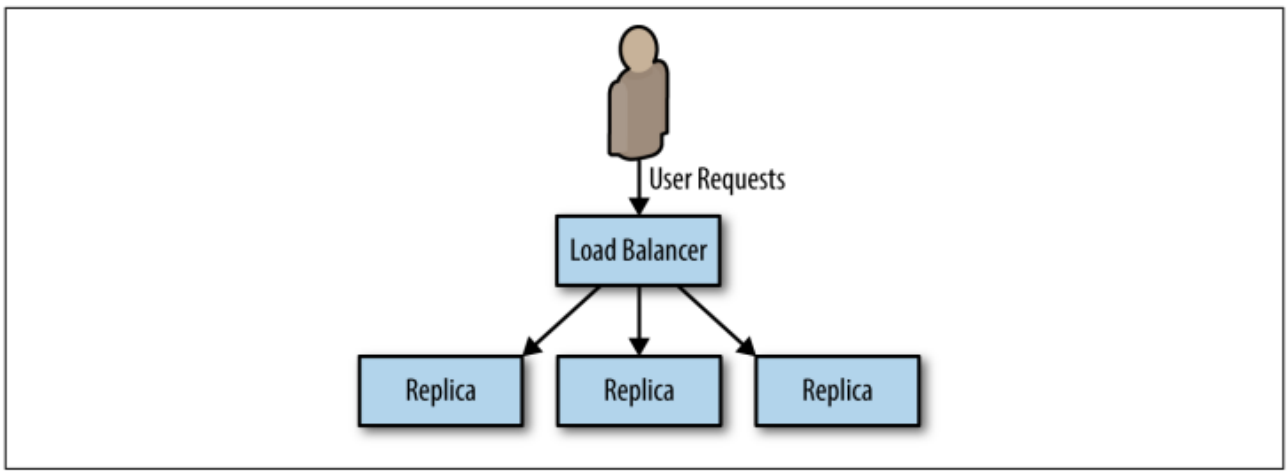

*Figure II-1. A monolithic service with all functions in a single container*




b. Replicated Load-Balanced Services

# Stateless Services

Stateless services are ones that don't require saved state to operate correctly. In the simplest stateless applications, even individual requests may be routed to separate instances of the service (see Figure 5-1). Examples of stateless services include things like static content servers and complex middleware systems that receive and aggregate responses from numerous different backend systems.

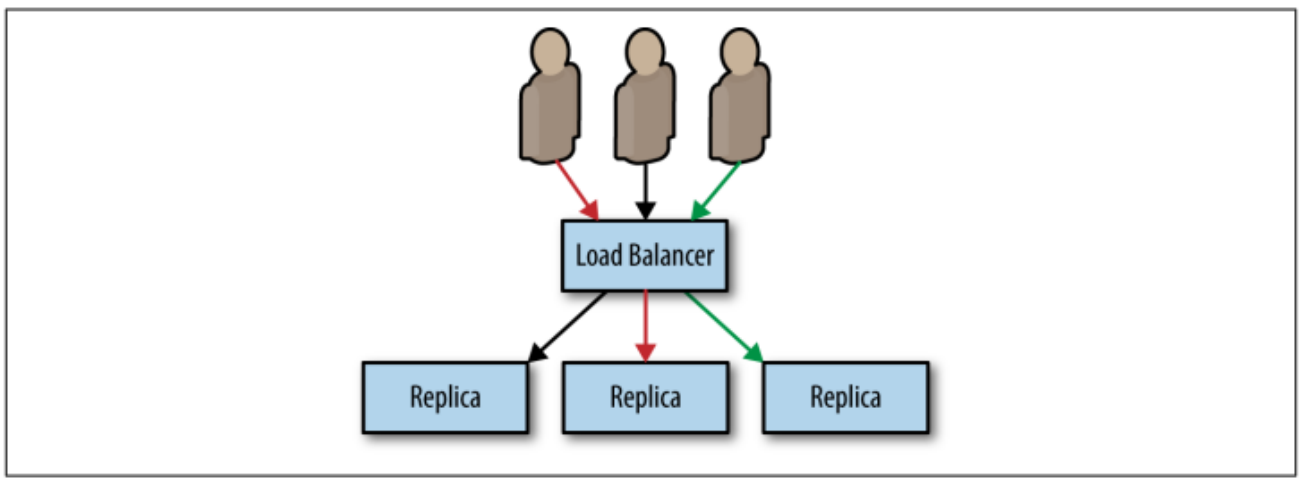


## Readiness Probes for Load Balancing

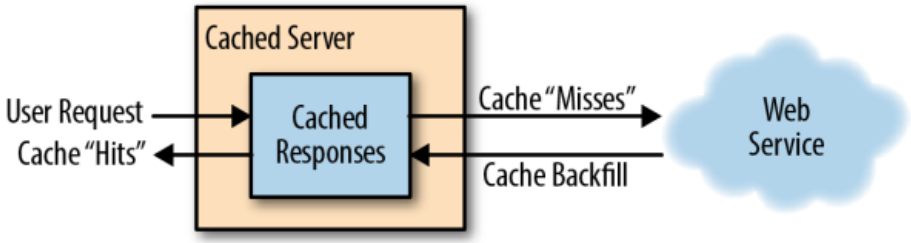
Of course, simply replicating your service and adding a load balancer is only part of a complete pattern for stateless replicated serving. When designing a replicated service, it is equally important to build and deploy a readiness probe to inform the load balancer. We have discussed how health probes can be used by a container orchestration system to determine when an application needs to be restarted. In contrast, a *readiness probe* determines when an application is ready to serve user requests. The reason for the differentiation is that many applications require some time to become initialized before they are ready to serve. They may need to connect to databases, load plugins, or download serving files from the network. In all of these cases, the containers are *alive*, but they are not *ready*. When building an application for a replicated service pattern, be sure to include a special URL that implements this readiness check.

# Session Tracked Services

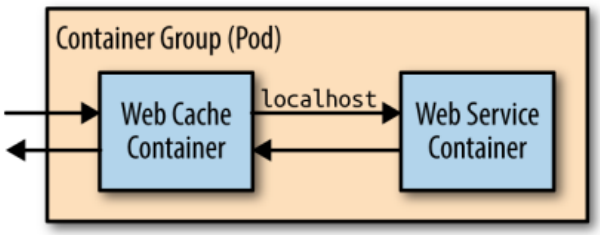
The previous examples of the stateless replicated pattern routed requests from all users to all replicas of a service. While this ensures an even distribution of load and fault tolerance, it is not always the preferred solution. Often there are reasons for wanting to ensure that a particular user's requests always end up on the same machine. Sometimes this is because you are caching that user's data in memory, so landing on the same machine ensures a higher cache hit rate. Sometimes it is because the interaction is long-running in nature, so some amount of state is maintained between requests. Regardless of the reason, an adaption of the stateless replicated service pattern is to use session tracked services, which ensure that all requests for a single user map to the same replica, as illustrated in Figure 5-3.



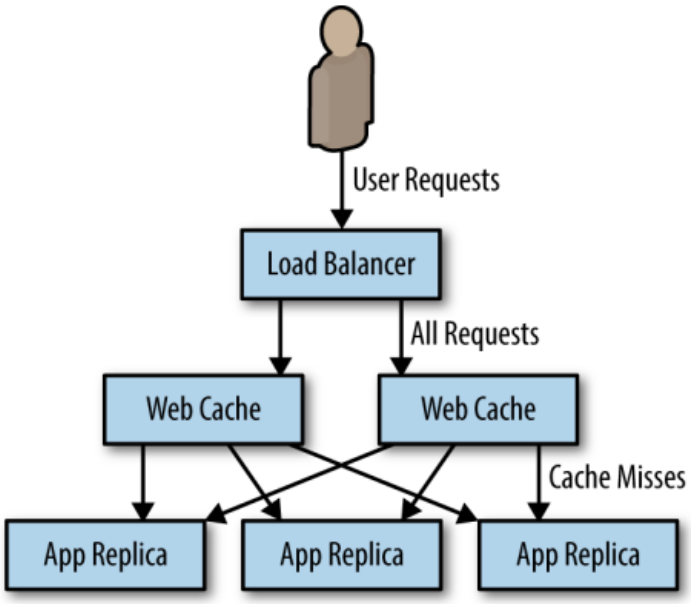

Example 1 Caching layer




The simplest way to deploy the web cache is alongside each instance of your web server using the sidecar pattern (see Figure 5-5).




Though this approach is simple, it has some disadvantages, namely that you will have to scale your cache at the same scale as your web servers. This is often not the approach you want. For your cache, you want as few replicas as possible with lots of resources for each replica (e.g., rather than 10 replicas with 1 GB of RAM each, you'd want two replicas with 5 GB of RAM each). To understand why this is preferable, consider that every page will be stored in every replica. With 10 replicas, you will store every page 10 times, reducing the overall set of pages that you can keep in memory in the cache. This causes a reduction in the *hit rate*, the fraction of the time that a request can be served out of cache, which in turn decreases the utility of the cache. Though you do want a few large caches, you might also want lots of small replicas of your web servers. Many languages (e.g., NodeJS) can really only utilize a single core, and thus you want many replicas to be able to take advantages of multiple cores, even on the same machine. Therefore, it makes the most sense to configure your caching layer as a second stateless replicated serving tier above your web-serving tier, as illustrated in Figure 5-6.




Unless you are careful, caching can break session tracking. The reason for this is that if you use default IP address affinity and load balancing, all requests will be sent from the IP addresses of the cache, not the end user of your service. If you've followed the advice previously given and deployed a few large caches, your IP-address-based affinity may in fact mean that some replicas of your web layer see *no* traffic. Instead, you need to use something like a cookie or HTTP header for session tracking.

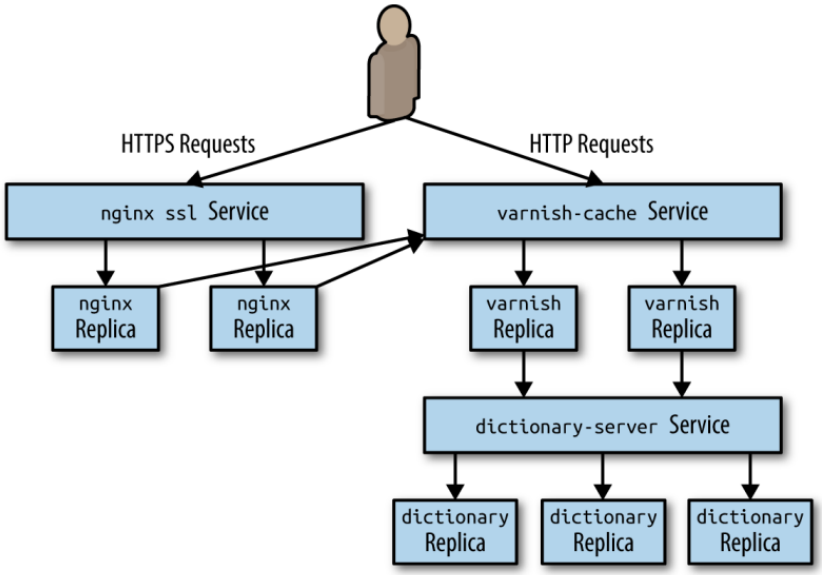
Few of us build sites with the expectation that we will encounter a denial-of-service attack. But as more and more of us build APIs, a denial of service can come simply from a developer misconfiguring a client or a site-reliability engineer accidentally running a load test against a production installation. Thus, it makes sense to add general denial-of-service defense via rate limiting to the caching layer. Most HTTP reverse proxies like Varnish have capabilities along this line. In particular, Varnish has a `throttle` module that can be configured to provide throttling based on IP address and request path, as well as whether or not a user is logged in.

If you are deploying an API, it is generally a best practice to have a relatively small rate limit for anonymous access and then force users to log in to obtain a higher rate limit. Requiring a login provides auditing to determine who is responsible for the unexpected load, and also offers a barrier to would-be attackers who need to obtain multiple identities to launch a successful attack.

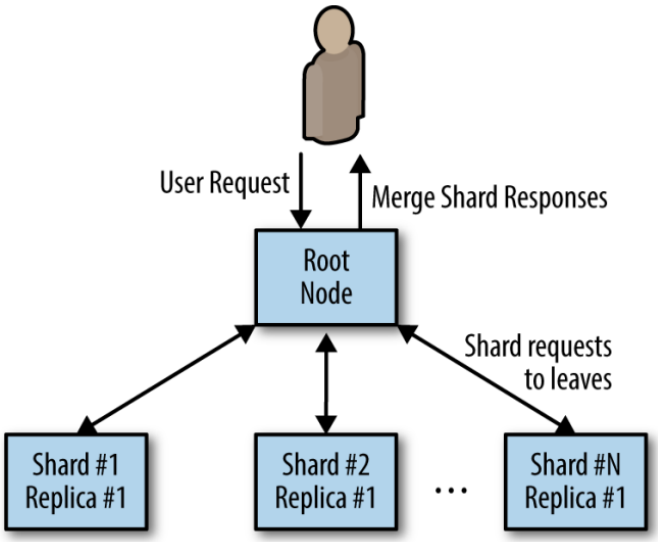
When a user hits the rate limit, the server will return the `429 error code` indicating that too many requests have been issued. However, many users want to understand how many requests they have left before hitting that limit. To that end, you will likely also want to populate an HTTP header with the remaining-calls information. Though there isn't a standard header for returning this data, many APIs return some variation of `X-RateLimit-Remaining`.

Example 2 SSL Termination

In addition to performing caching for performance, one of the other common tasks performed by the edge layer is SSL termination. Even if you plan on using SSL for communication between layers in your cluster, you should still use different certificates for the edge and your internal services. Indeed, each individual internal service should use its own certificate to ensure that each layer can be rolled out independently. Unfortunately, the Varnish web cache can't be used for SSL termination, but fortunately, the `nginx` application can. Thus we want to add a third layer to our stateless application pattern, which will be a replicated layer of `nginx` servers that will handle SSL termination for HTTPS traffic and forward traffic on to our Varnish cache. HTTP traffic continues to travel to the Varnish web cache, and Varnish forwards traffic on to our web application, as shown in Figure 5-8.

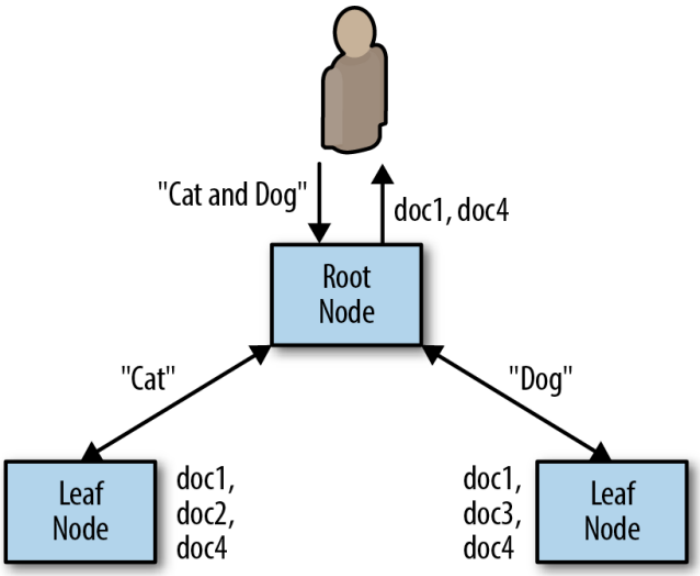



d. Scatter/Gather




To understand this in more concrete terms, imagine that you need to service a user request *R* and it takes one minute for a single core to produce the answer *A* to this request. If we program a multi-threaded application, we can parallelize this request on a single machine by using multiple cores. Given this approach and a 30 core processor (yes, typically it would be a 32 core processor, but 30 makes the math cleaner), we can reduce the time that it takes to process a single request down to 2 seconds (60 seconds of computation split across 30 threads for computation is equal to 2 seconds). But even two seconds is pretty slow to service a user's web request. Additionally, truly achieving a completely parallel speed up on a single process is going to be tricky as things like memory, network, or disk bandwidth start to become the bottleneck. Instead of parallelizing an application across cores on a single machine, we can use the scatter/gather pattern to parallelize requests across multiple processes on many different machines. In this way, we can improve our overall latency requests,

Example 1 Distributed Document Search




Together, these complications of scatter/gather systems lead us to some conclusions:

- Increased parallelism doesn't always speed things up because of overhead on each node.
- Increased parallelism doesn't always speed things up because of the straggler problem.
- The performance of the 99th percentile is more important than in other systems because each user request actually becomes numerous requests to the service.

### e. Functions and Event-Driven Processing

So far, we have examined design for systems with long-running computation. The servers that handle user requests are always up and running. This pattern is the right one for many applications that are under heavy load, keep a large amount of data in memory, or require some sort of background processing. However, there is a class of applications that might only need to temporarily come into existence to handle a single request, or simply need to respond to a specific event. This style of request or event-driven application design has flourished recently as large-scale public cloud providers have developed *function-as-a-service* (FaaS) products. More recently, FaaS implementations have also emerged running on top of cluster orchestrators in private cloud or physical environments. This chapter describes emerging architectures for this new style of computing. In many cases, FaaS is a component in a broader architecture rather than a complete solution.

Oftentimes, FaaS is referred to as *serverless* computing. And while this is true (you don't see the servers in FaaS) it's worth differentiating between event-driven FaaS and the broader notion of serverless computing. Indeed, serverless computing can apply to a wide variety of computing services; for example, a multi-tenant container orchestrator (container-as-a-service) is serverless but not event-driven. Conversely, an open source FaaS running on a cluster of physical machines that you own and administer is event-driven but not serverless. Understanding this distinction enables you to determine when event-driven, serverless, or both is the right choice for your application.

## The Benefits of FaaS

The benefits of FaaS are primarily for the developer. It dramatically simplifies the distance from code to running service. Because there is no artifact to create or push beyond the source code itself, FaaS makes it simple to go from code on a laptop or web browser to running code in the cloud.

Likewise, the code that is deployed is managed and scaled automatically. As more traffic is loaded onto the service, more instances of the function are created to handle that increase in traffic. If a function fails due to application or machine failures, it is automatically restarted on some other machine.

Finally, much like containers, functions are an even more granular building block for designing distributed systems. Functions are stateless and thus any system you build on top of functions is inherently more modular and decoupled than a similar system built into a single binary. But, of course, this is also the challenge of developing systems in FaaS. The decoupling is both a strength and a weakness. The following section describes some of the challenges that come from developing systems using FaaS.

## The Challenges of FaaS

As described in the previous section, developing systems using FaaS forces you to strongly decouple each piece of your service. Each function is entirely independent. The only communication is across the network, and each function instance cannot have local memory, requiring all states to be stored in a storage service. This forced decoupling can improve the agility and speed with which you can develop services, but it can also significantly complicate the operations of the same service.

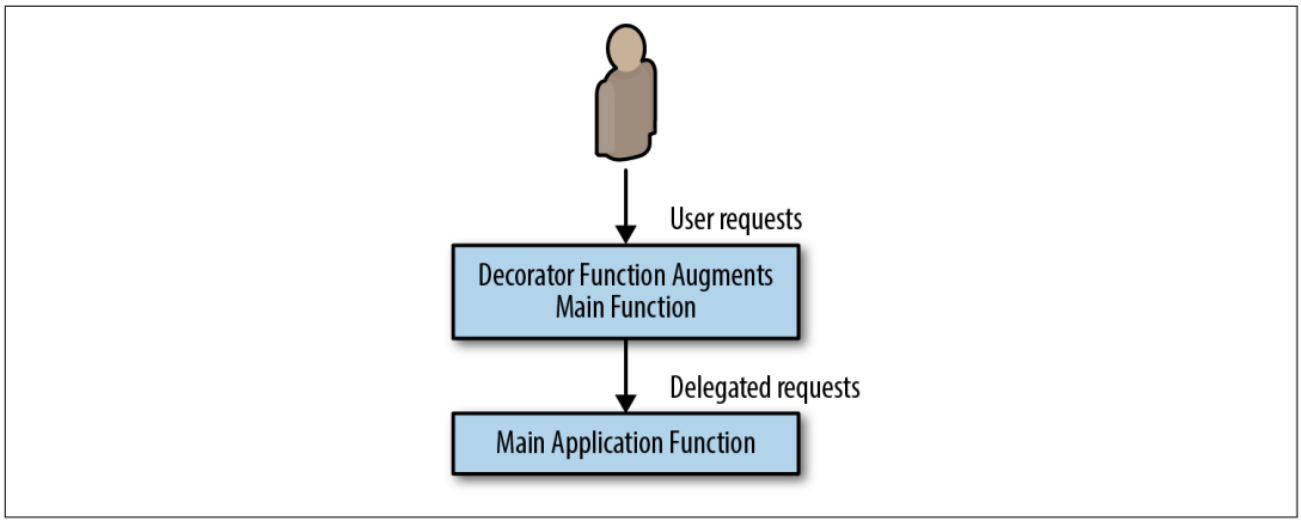
In particular, it is often quite difficult to obtain a comprehensive view of your service, determine how the various functions integrate with one another, and understand when things go wrong, and why they go wrong. Additionally, the request-based and serverless nature of functions means that certain problems are quite difficult to detect. As an example, consider the following functions:

- *`functionA()`* which calls *`functionB()`*
- *`functionB()`* which calls *`functionC()`*
- *`functionC()`* which calls back to *`functionA()`*

Now consider what happens when a request comes into any of these functions: it kicks off an infinite loop that only terminates when the original request times out (and possibly not even then) or when you run out of money to pay for requests in the system. Obviously, the above example is quite contrived, but it is actually quite difficult to detect in your code. Since each function is radically decoupled from the other functions, there is no real representation of the dependencies or interactions between different functions. These problems are not unsolvable, and I expect that as FaaSs mature, more analysis and debugging tools will provide a richer experience to understand how and why an application comprised of FaaS is performing the way that it does.

Example 1 Decorator Pattern: Request or Response Transformation

FaaS is ideal for deploying simple functions that can take an input, transform it into an output, and then pass it on to a different service. This general pattern can be used to augment or decorate HTTP requests to or from a different service. A basic illustration of this pattern is shown in Figure 8-1.




Event vs microservices:

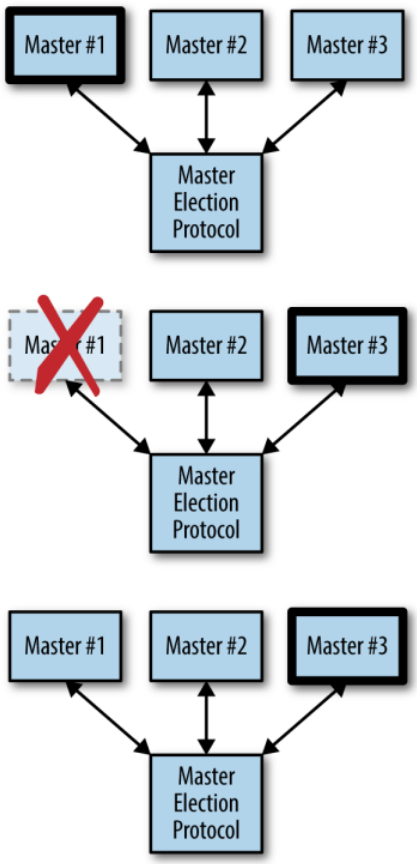
There are some applications that are inherently easier to think about in terms of a pipeline of decoupled events. These event pipelines often resemble the flowcharts of old. They can be represented as a directed graph of connected event syncs. In the event pipeline pattern, each node is a different function or webhook, and the edges linking the graph together are HTTP or other network calls to the function/webhook. In general, there is no shared state between the different pieces of the pipeline, but there may be a context or other reference point that can be used to look up information in shared storage.

So what is the difference between this type of pipeline and a "microservices" architecture? There are two central differences. The first is the main difference between functions in general and long-running services, which is that an event-based pipeline is by its very nature event-driven. Conversely, a microservices architecture features a collection of long-running services. Additionally, event-driven pipelines may be highly asynchronous and diverse in the things that they connect together. For example, while it is difficult to see how a human approving a ticket in a ticketing system like Jira could be integrated into a microservices application, it's quite easy to see how that event could be incorporated into a event-driven pipeline.

f. Ownership election

The previous patterns that we have seen have been about distributing requests in order to scale requests per second, the state being served, or the time to process a request. This final chapter on multi-node serving patterns is about how you scale assignment. In many different systems, there is a notion of *ownership* where a specific process owns a specific task. We have previously seen this in the context of sharded and hot-sharded systems where specific instances owned specific sections of the sharded key space.

A general diagram of distributed ownership is shown in Figure 9-1. In the diagram, there are three replicas that could be the owner or master. Initially, the first replica is the master. Then that replica fails, and replica number three then becomes the master. Finally, replica number one recovers and returns to the group, but replica three remains as the master/owner.




Assuming that you run your singleton in a container orchestration system like Kubernetes, you have the following guarantees:

- If the container crashes, it will automatically be restarted

- If the container hangs, and you implement a health check, it will automatically be restarted
- If the machine fails, the container will be moved to a different machine

Master election:

There are two ways to implement this master election. This first is to implement a distributed consensus algorithm like Paxos or RAFT, but the complexity of these algorithms make them beyond the scope of this book and not worthwhile to implement. Implementing one of these algorithms is akin to implementing locks on top of assembly code compare-and-swap instructions. It's an interesting exercise for an undergraduate computer science course, but it is not something that is generally worth doing in practice.

Fortunately, there are a large number of distributed key-value stores that have implemented such consensus algorithms for you. At a general level, these systems provide a replicated, reliable data store and the primitives necessary to build more complicated locking and election abstractions on top. Examples of these distributed stores include etcd, ZooKeeper, and consul. The basic primitives that these systems provide is the ability to perform a compare-and-swap operation for a particular key. If you haven't seen compare-and-swap before, it is basically an atomic operation that looks like this:

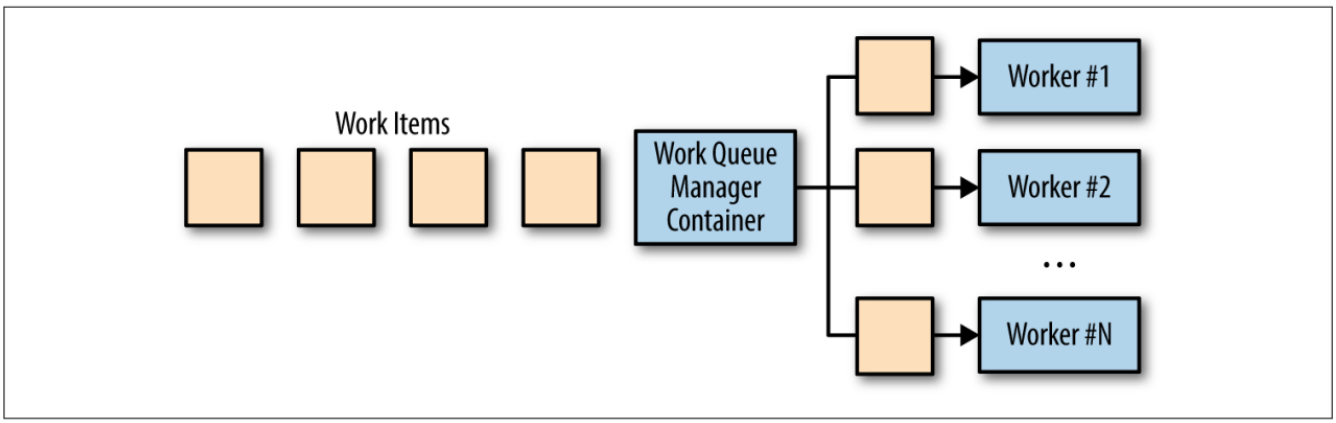
## Implementing Ownership

While locks are great for establishing temporary ownership of some critical component, sometimes you want to take ownership for the duration of the time that the component is running. For example, in a highly available deployment of Kubernetes, there are multiple replicas of the scheduler but only one replica is actively making scheduling decisions. Further, once it becomes the active scheduler, it remains the active scheduler until that process fails for some reason.

## 1.3 Batch Computational Patterns

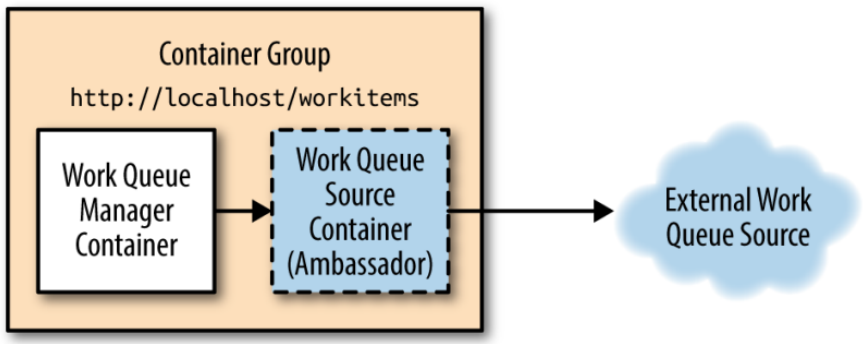
### a. Work Queue Systems

The simplest form of batch processing is a *work queue*. In a work queue system, there is a batch of work to be performed. Each piece of work is wholly independent of the other and can be processed without any interactions. Generally, the goals of the work queue system are to ensure that each piece of work is processed within a certain amount of time. Workers are scaled up or scaled down to ensure that the work can be handled. An illustration of a generic work queue is shown in Figure 10-1.

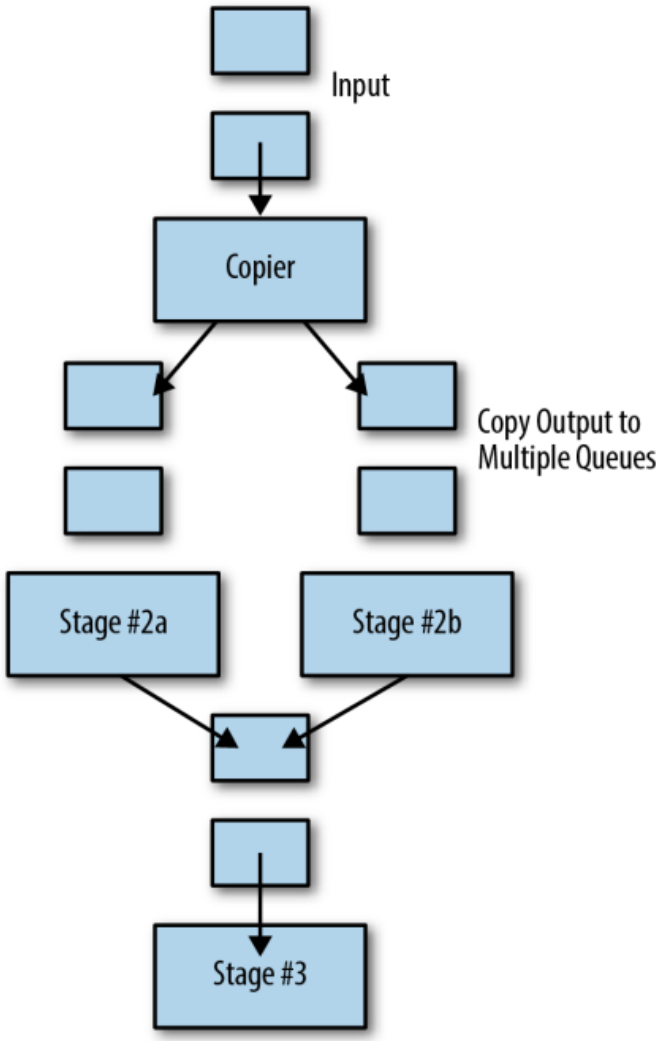



## A Generic Work Queue System

The work queue is an ideal way to demonstrate the power of distributed system patterns. Most of the logic in the work queue is wholly independent of the actual work being done, and in many cases the delivery of the work can be performed in an independent manner as well. To illustrate this point, consider the work queue illustrated in Figure 10-1. If we take a look at it again and identify functionality that can be provided by a shared set of *library containers*, it becomes apparent that most of the implementation of a containerized work queue can be shared across a wide variety of users, as shown in Figure 10-2.



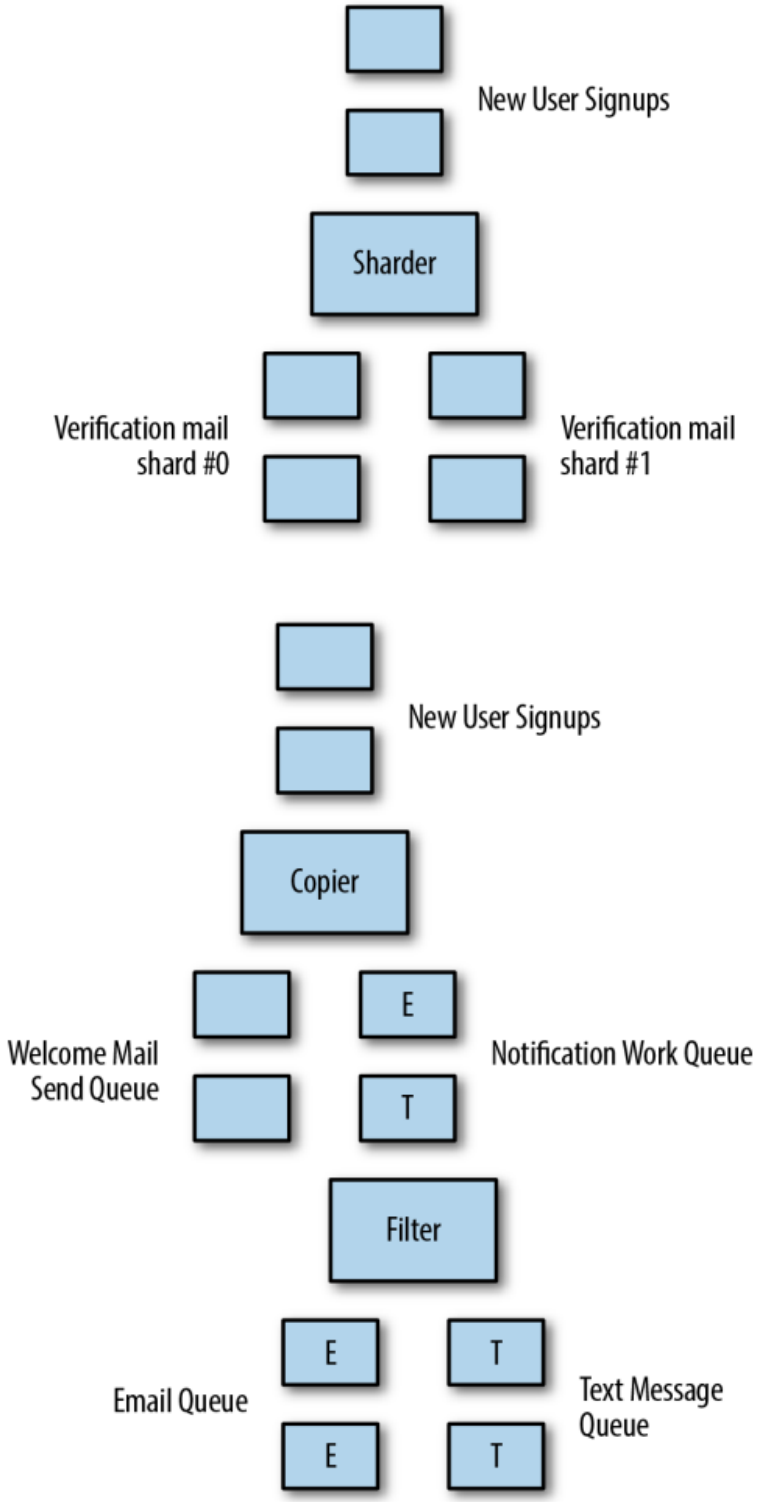

b. Event-Driven Batch Processing




**Patterns of Event-Driven Processing**

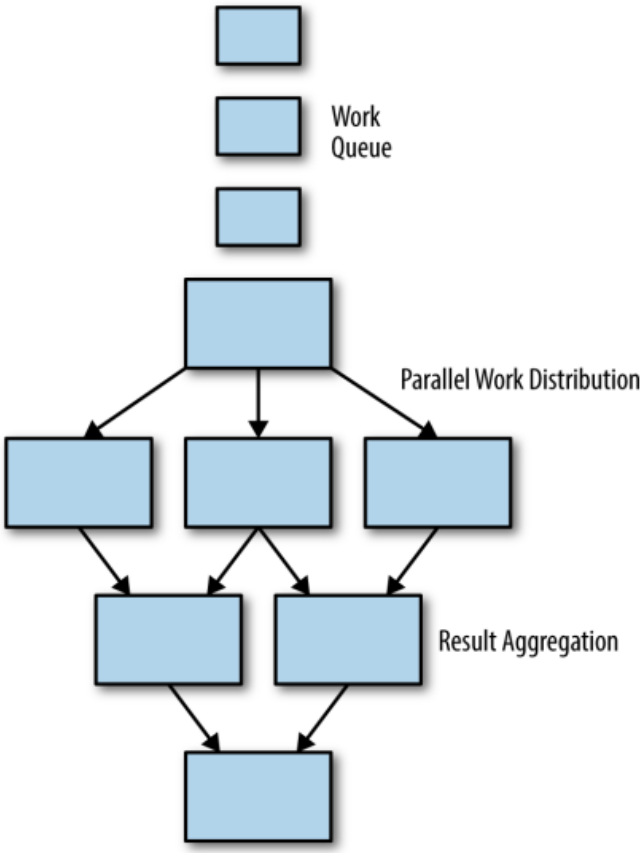
- Copier
- Fitler
- Spliter
- Sharder
- Merger

**Example 1 Hands On: Building an Event-Driven Flow for New User Sign-Up**



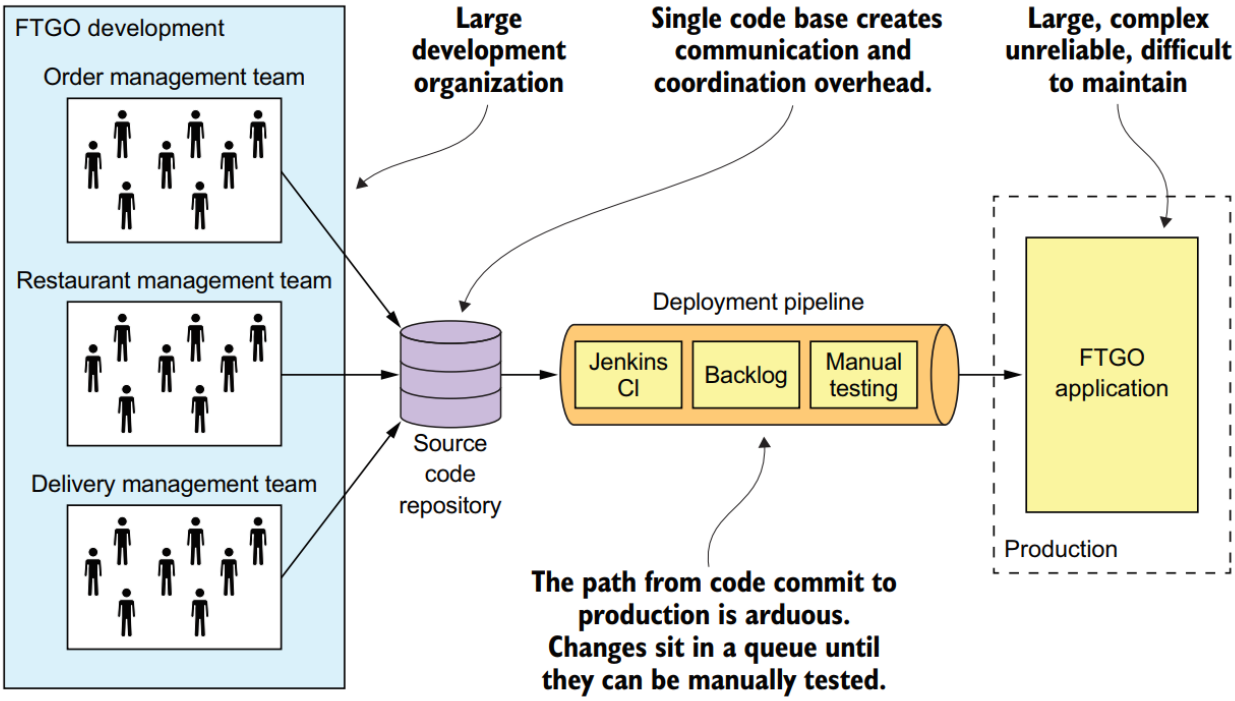

c. Coordinated Batch Processing

The previous chapter described a number of patterns for splitting and chaining queues together to achieve more complex batch processing. Duplicating and producing multiple different outputs is often an important part of batch processing, but sometimes it is equally important to pull multiple outputs back together in order to generate some sort of aggregate output. A generic illustration of such a pattern is shown in Figure 12-1.




## 2. Microservices patterns

### 2.1 Monolith Issue




### DEVELOPMENT IS SLOW

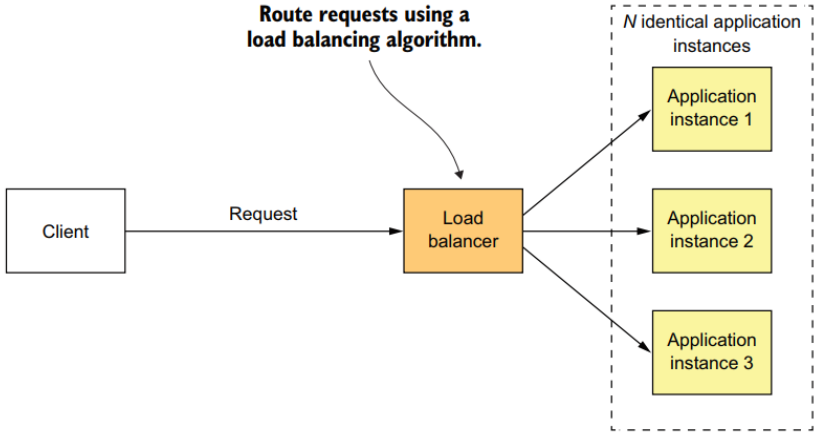
As well as having to fight overwhelming complexity, FTGO developers find day-to-day development tasks slow. The large application overloads and slows down a developer's IDE. Building the FTGO application takes a long time. Moreover, because it's so large, the application takes a long time to start up. As a result, the edit-build-run-test loop takes a long time, which badly impacts productivity.

## Microservice definition

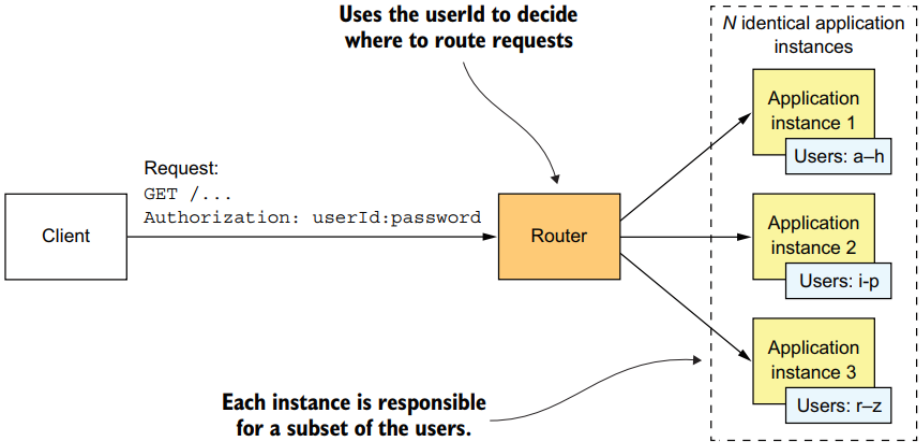
Today, the growing consensus is that if you're building a large, complex application, you should consider using the microservice architecture. But what are *microservices* exactly? Unfortunately, the name doesn't help because it overemphasizes size. There are numerous definitions of the microservice architecture. Some take the name too literally and claim that a service should be tiny—for example, 100 LOC. Others claim that a service should only take two weeks to develop. Adrian Cockcroft, formerly of Netflix, defines a microservice architecture as a service-oriented architecture composed of loosely coupled elements that have bounded contexts. That's not a bad definition, but it is a little dense. Let's see if we can do better.

## Scaling Microservice

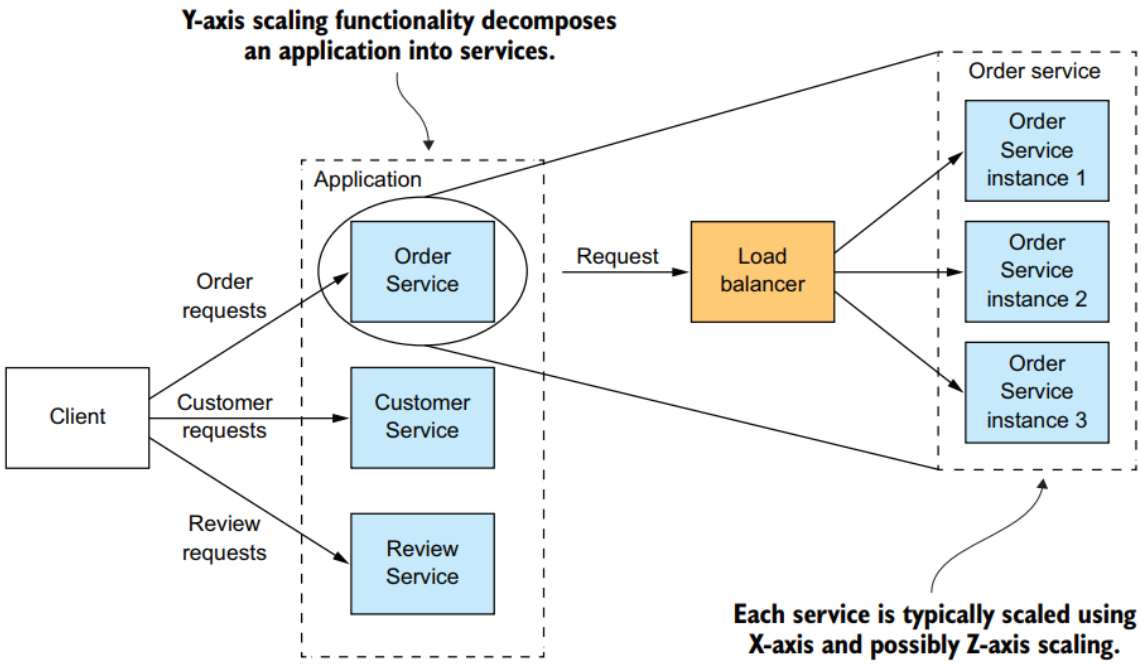
X-axis


Route requests using a load balancing algorithm.
N identical application instances
Application instance 1
Application instance 2
Application instance 3
Client
Request
Load balancer

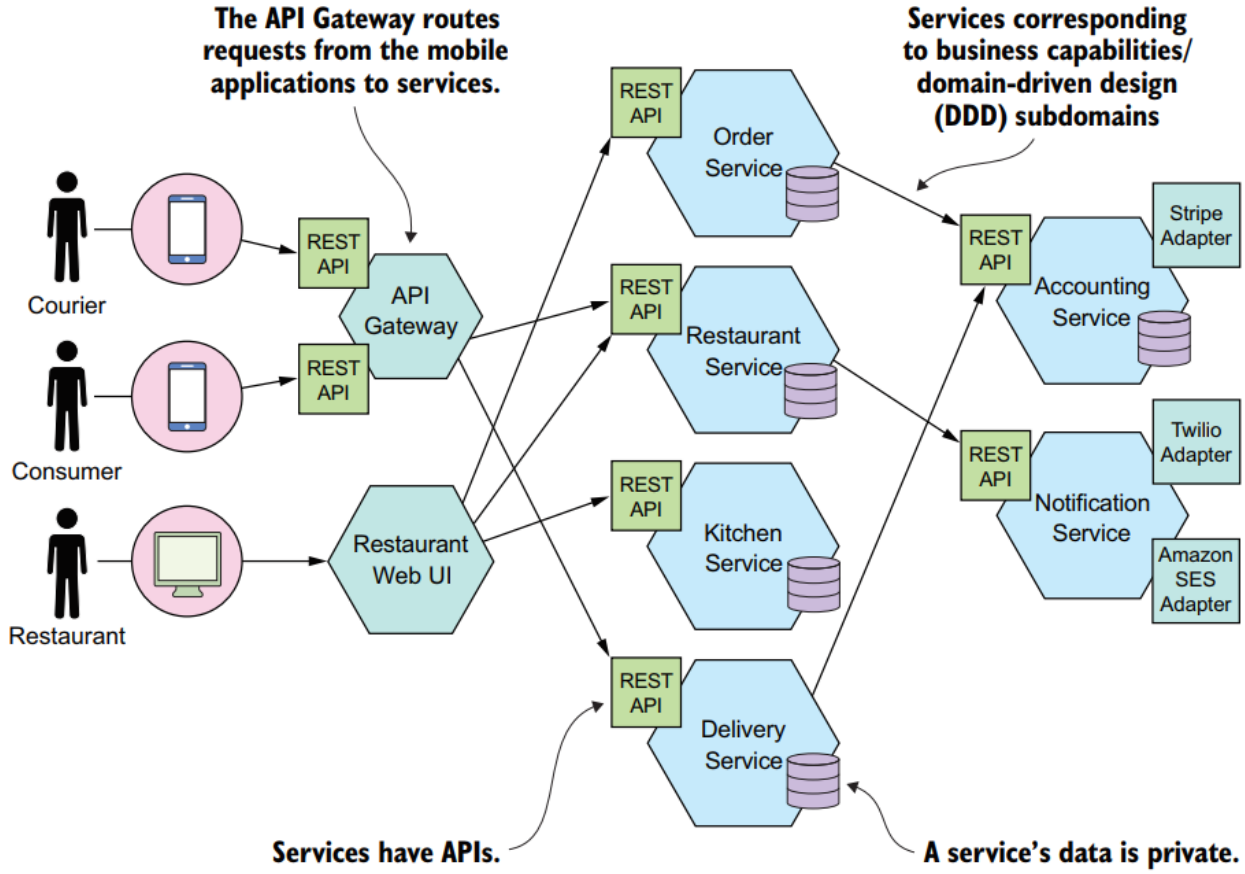
Z-axis


Uses the userId to decide where to route requests
N identical application instances
Application instance 1
Users: a–h
Application instance 2
Users: i-p
Application instance 3
Users: r–z
Request:
GET /...
Authorization: userId:password
Client
Router
Each instance is responsible for a subset of the users.

Y-axis


Y-axis scaling functionality decomposes an application into services.
Order service
Order Service instance 1
Order Service instance 2
Order Service instance 3
Application
Order Service
Request
Load balancer
Order requests
Client
Customer requests
Customer Service
Review requests
Review Service
Each service is typically scaled using X-axis and possibly Z-axis scaling.

## The FTGO microservice architecture



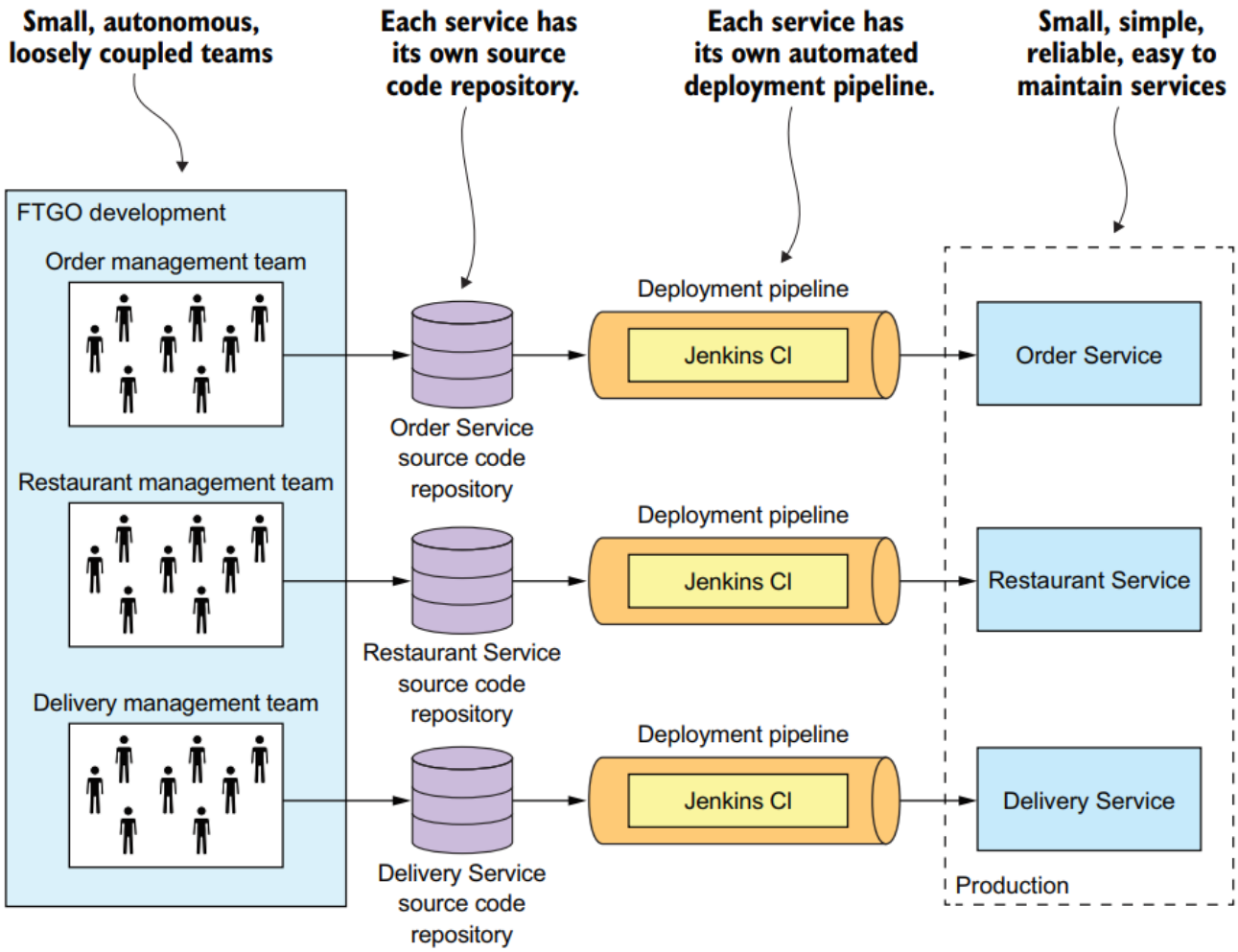

## Comparing the microservice architecture and SOA

| | SOA | Microservices |
|---|---|---|
| Inter-service communication | Smart pipes, such as Enterprise Service Bus, using heavyweight protocols, such as SOAP and the other WS* standards. | Dumb pipes, such as a message broker, or direct service-to-service communication, using lightweight protocols such as REST or gRPC |
| Data | Global data model and shared databases | Data model and database per service |
| Typical service | Larger monolithic application | Smaller service |

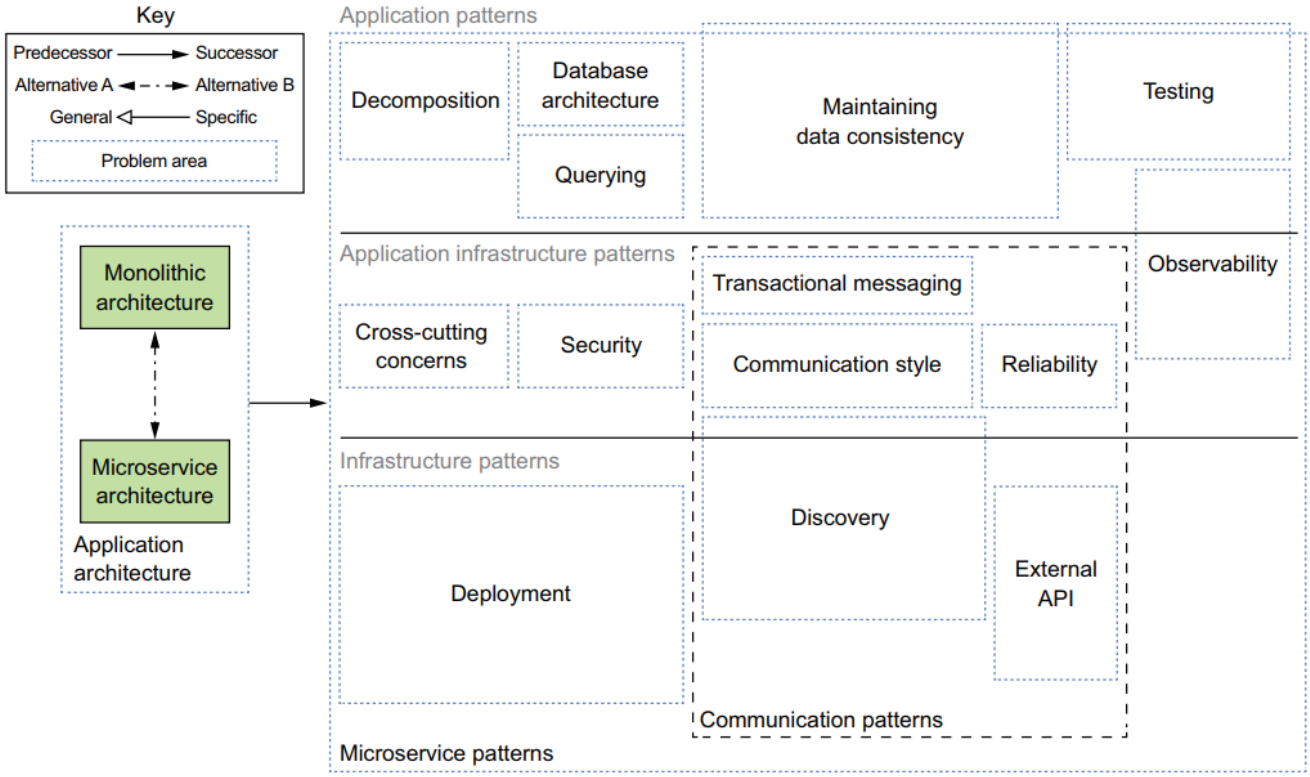
SOA and the microservice architecture usually use different technology stacks. SOA applications typically use heavyweight technologies such as SOAP and other WS* standards. They often use an ESB, a *smart pipe* that contains business and message-processing logic to integrate the services. Applications built using the microservice architecture tend to use lightweight, open source technologies. The services communicate via *dumb pipes*, such as message brokers or lightweight protocols like REST or gRPC.

SOA and the microservice architecture also differ in how they treat data. SOA applications typically have a global data model and share databases. In contrast, as mentioned earlier, in the microservice architecture each service has its own database. Moreover, as described in chapter 2, each service is usually considered to have its own domain model.

## Benefits and drawbacks of the microservice architecture




## Microservice architecture pattern language




## PATTERNS FOR DECOMPOSING AN APPLICATION INTO SERVICES

Deciding how to decompose a system into a set of services is very much an art, but there are a number of strategies that can help. The two decomposition patterns shown in figure 1.11 are different strategies you can use to define your application's architecture.

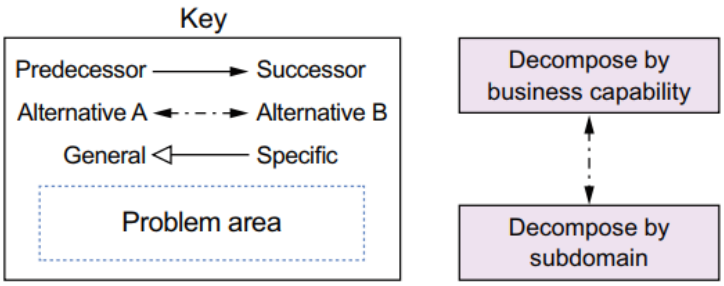
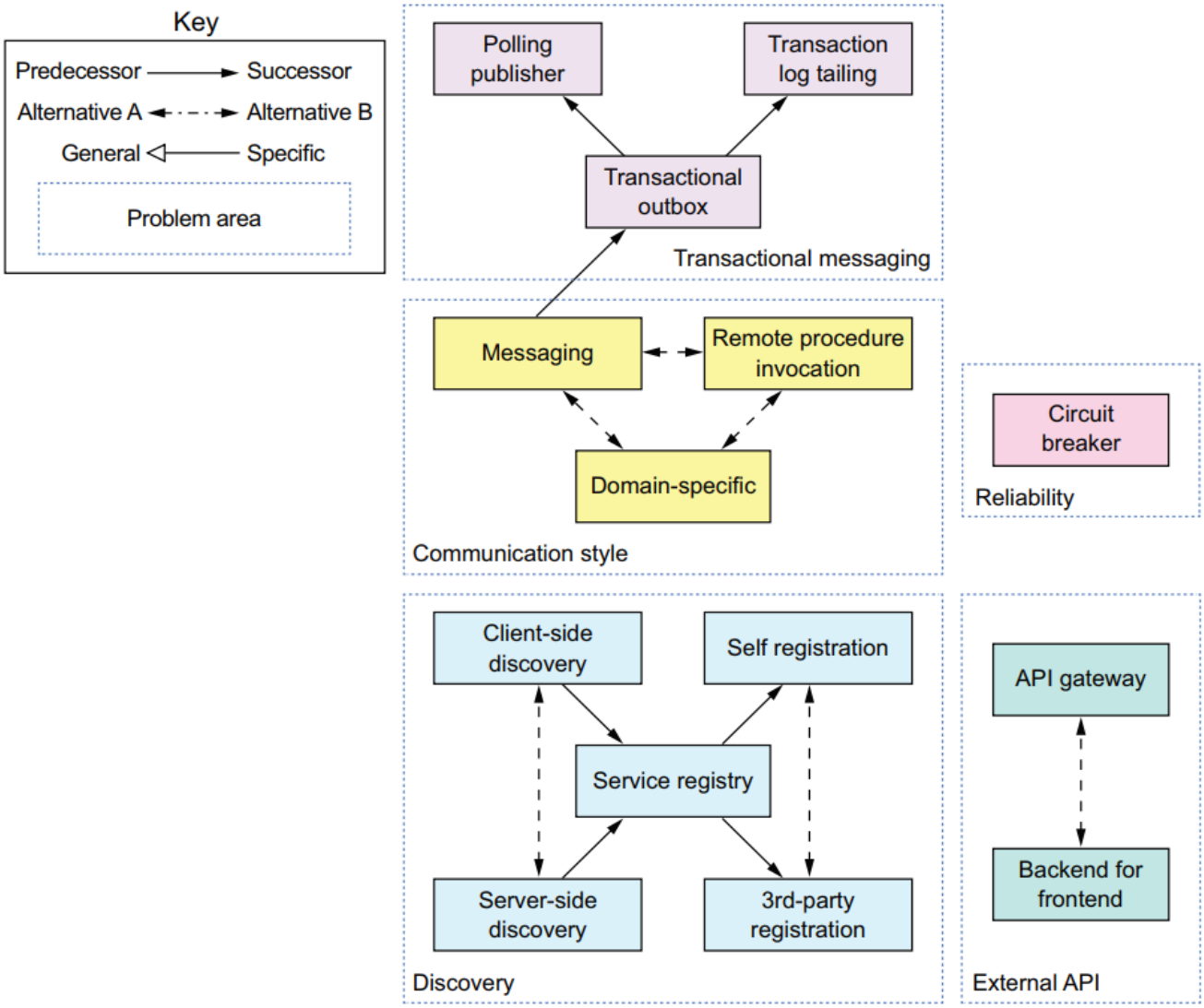
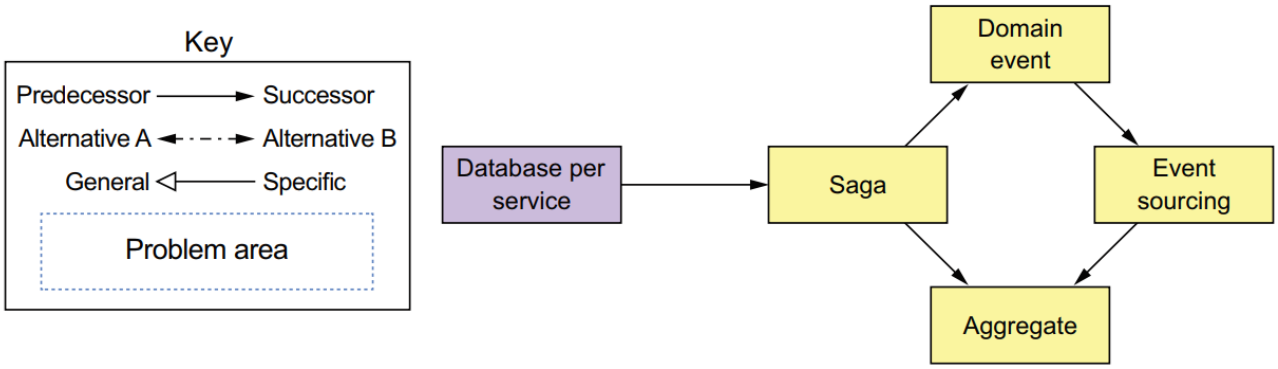



**Figure 1.11 There are two decomposition patterns: Decompose by business capability, which organizes services around business capabilities, and Decompose by subdomain, which organizes services around domain-driven design (DDD) subdomains.**



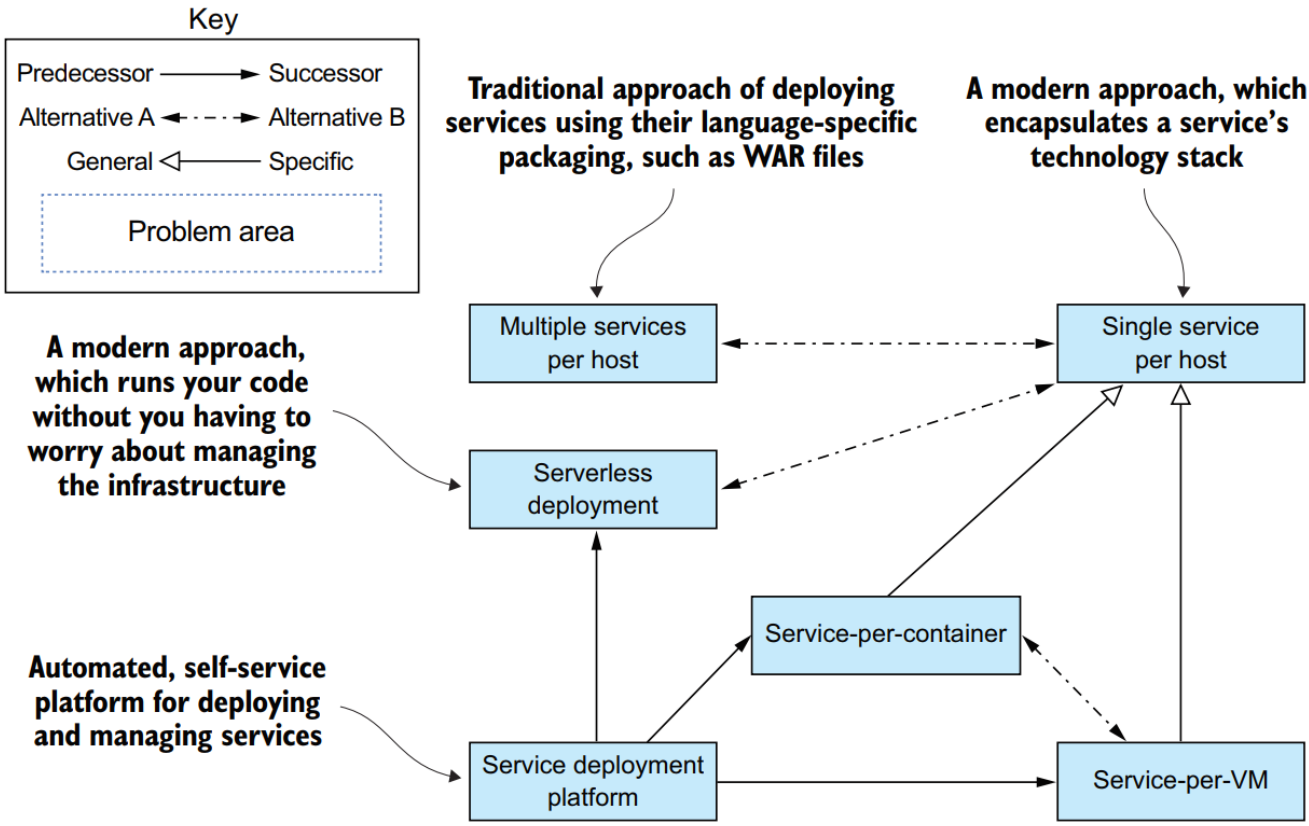

### DATA CONSISTENCY PATTERNS FOR IMPLEMENTING TRANSACTION MANAGEMENT

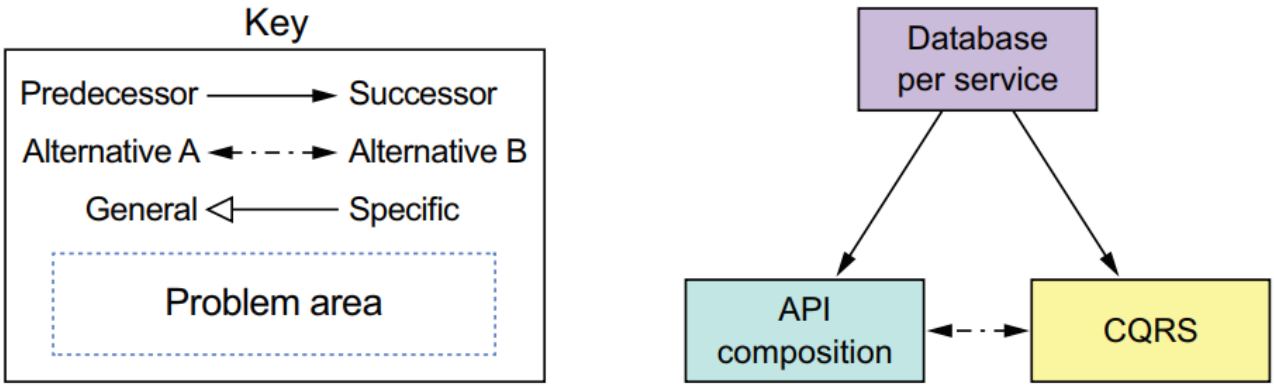
As mentioned earlier, in order to ensure loose coupling, each service has its own database. Unfortunately, having a database per service introduces some significant issues. I describe in chapter 4 that the traditional approach of using distributed transactions (2PC) isn't a viable option for a modern application. Instead, an application needs to maintain data consistency by using the Saga pattern. Figure 1.13 shows data-related patterns.

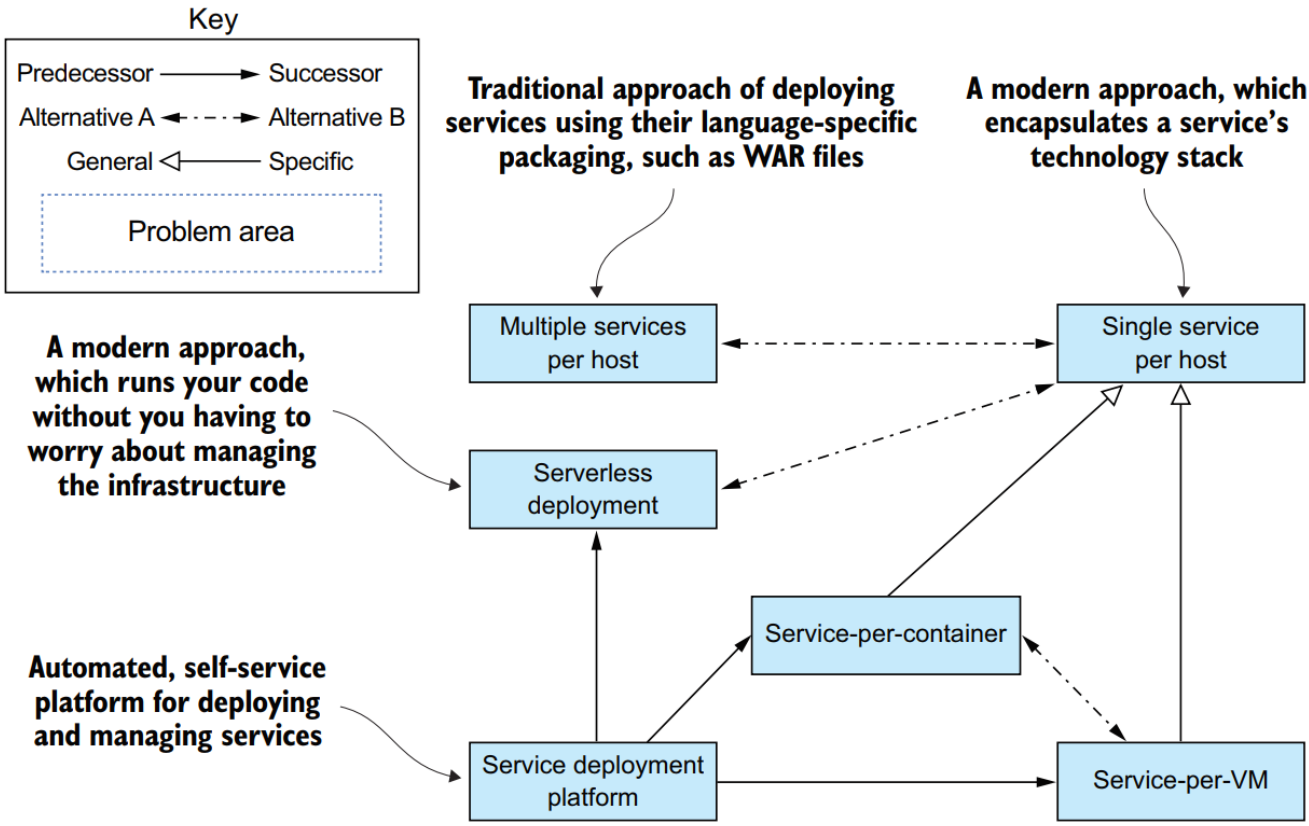



### PATTERNS FOR QUERYING DATA IN A MICROSERVICE ARCHITECTURE

The other issue with using a database per service is that some queries need to join data that's owned by multiple services. A service's data is only accessible via its API, so you can't use distributed queries against its database. Figure 1.14 shows a couple of patterns you can use to implement queries.


Key
Predecessor
Successor
Alternative A
Alternative B
General
Specific
Problem area
Traditional approach of deploying services using their language-specific packaging, such as WAR files
A modern approach, which encapsulates a service's technology stack
Multiple services per host
Single service per host
A modern approach, which runs your code without you having to worry about managing the infrastructure
Serverless deployment
Service-per-container
Automated, self-service platform for deploying and managing services
Service deployment platform
Service-per-VM


Key
Predecessor
Successor
Alternative A
Alternative B
General
Specific
Problem area
Database per service
API composition
CQRS




## 2.2 Decomposition strategies

Overview of architectural styles

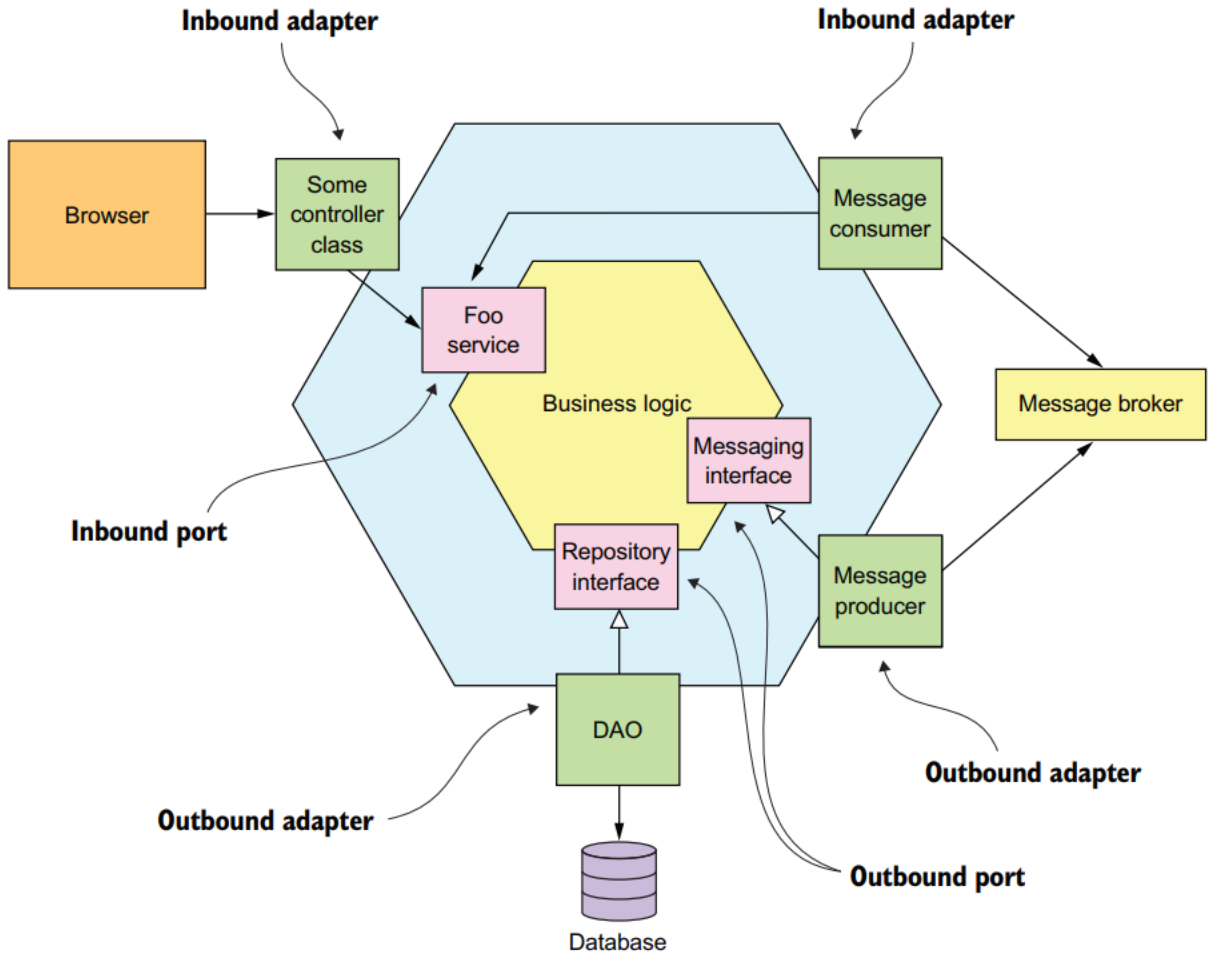
THE LAYERED ARCHITECTURAL STYLE

- **Presentation layer** —Contains code that implements the user interface or external APIs
- **Business logic layer**—Contains the business logic
- **Persistence layer**—Implements the logic of interacting with the database

THE HEXAGONAL ARCHITECTURE STYLE

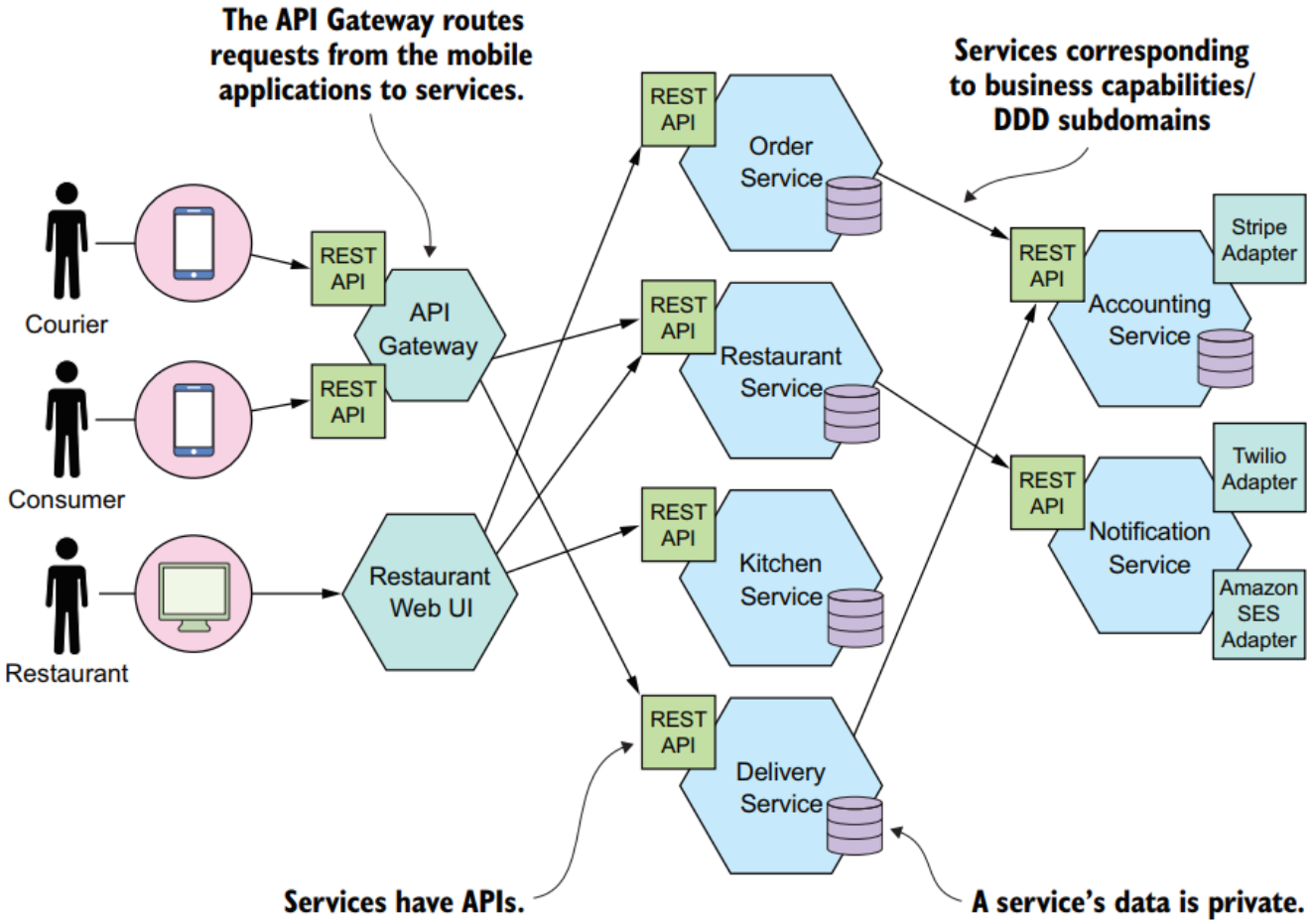
*Hexagonal architecture* is an alternative to the layered architectural style. As figure 2.2 shows, the hexagonal architecture style organizes the logical view in a way that places the business logic at the center. Instead of the presentation layer, the application has one or more *inbound adapters* that handle requests from the outside by invoking the business logic. Similarly, instead of a data persistence tier, the application has one or more *outbound adapters* that are invoked by the business logic and invoke external applications. A key characteristic and benefit of this architecture is that the business logic doesn't depend on the adapters. Instead, they depend upon it.

The business logic has one or more ports. A *port* defines a set of operations and is how the business logic interacts with what's outside of it. In Java, for example, a port is often a Java interface. There are two kinds of ports: inbound and outbound ports. An inbound port is an API exposed by the business logic, which enables it to be invoked by external applications. An example of an inbound port is a service interface, which defines a service's public methods. An outbound port is how the business logic invokes external systems. An example of an output port is a repository interface, which defines a collection of data access operations.



Surrounding the business logic are adapters. As with ports, there are two types of adapters: inbound and outbound. An inbound adapter handles requests from the outside world by invoking an inbound port. An example of an inbound adapter is a Spring MVC Controller that implements either a set of REST endpoints or a set of web pages. Another example is a message broker client that subscribes to messages. Multiple inbound adapters can invoke the same inbound port.

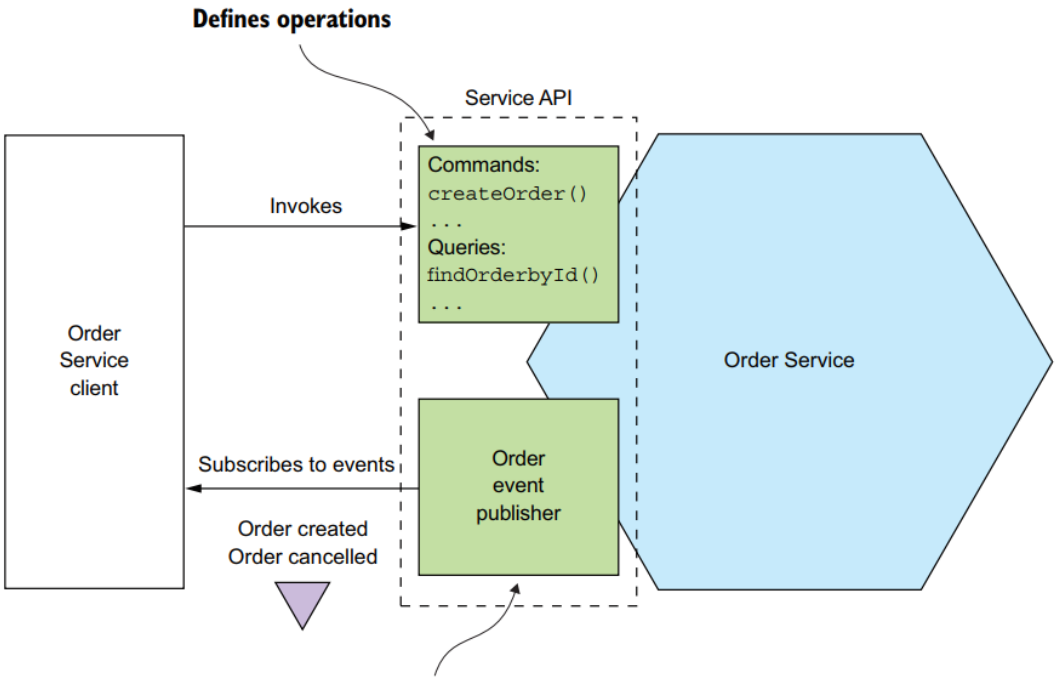
An outbound adapter implements an outbound port and handles requests from the business logic by invoking an external application or service. An example of an outbound adapter is a *data access object* (DAO) class that implements operations for accessing a database. Another example would be a proxy class that invokes a remote service. Outbound adapters can also publish events.



### WHAT IS A SERVICE?

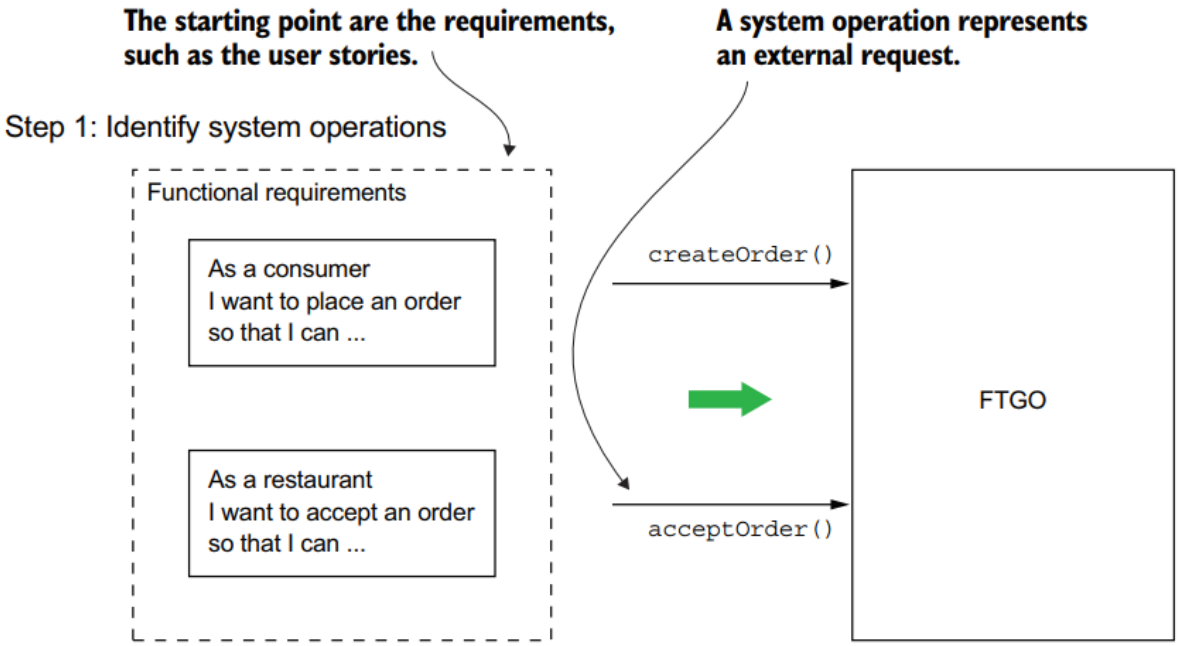
A *service* is a standalone, independently deployable software component that implements some useful functionality. Figure 2.4 shows the external view of a service, which in this example is the `Order Service`. A service has an API that provides its clients access to its functionality. There are two types of operations: commands and queries. The API consists of commands, queries, and events. A command, such as `createOrder()`, performs actions and updates data. A query, such as `findOrderById()`, retrieves data. A service also publishes events, such as `OrderCreated`, which are consumed by its clients.

A service's API encapsulates its internal implementation. Unlike in a monolith, a developer can't write code that bypasses its API. As a result, the microservice architecture enforces the application's modularity.

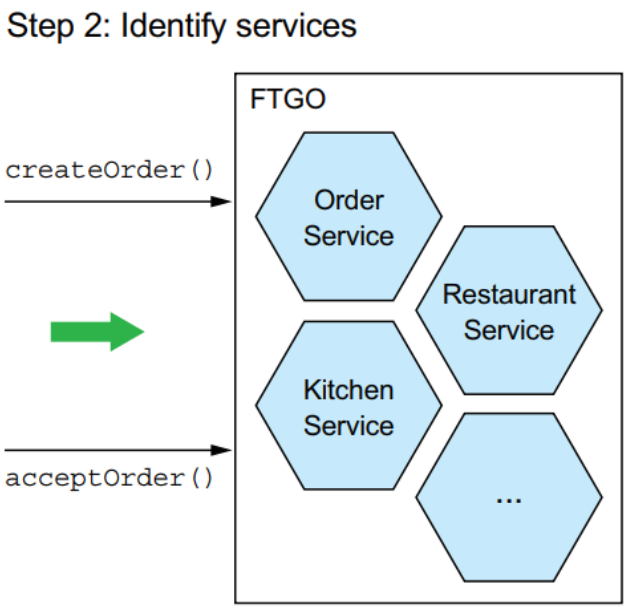



Defining an application's microservice architecture

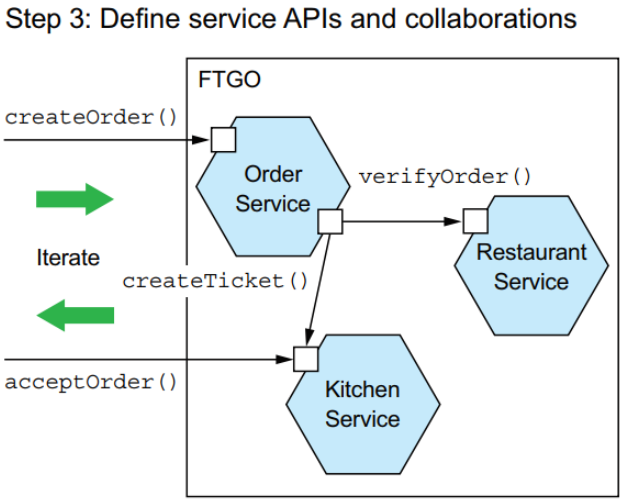
Step1


The starting point are the requirements, such as the user stories.
A system operation represents an external request.
Step 1: Identify system operations
Functional requirements
As a consumer
I want to place an order
so that I can ...
As a restaurant
I want to accept an order
so that I can ...
createOrder()
acceptOrder()
FTGO

Step2

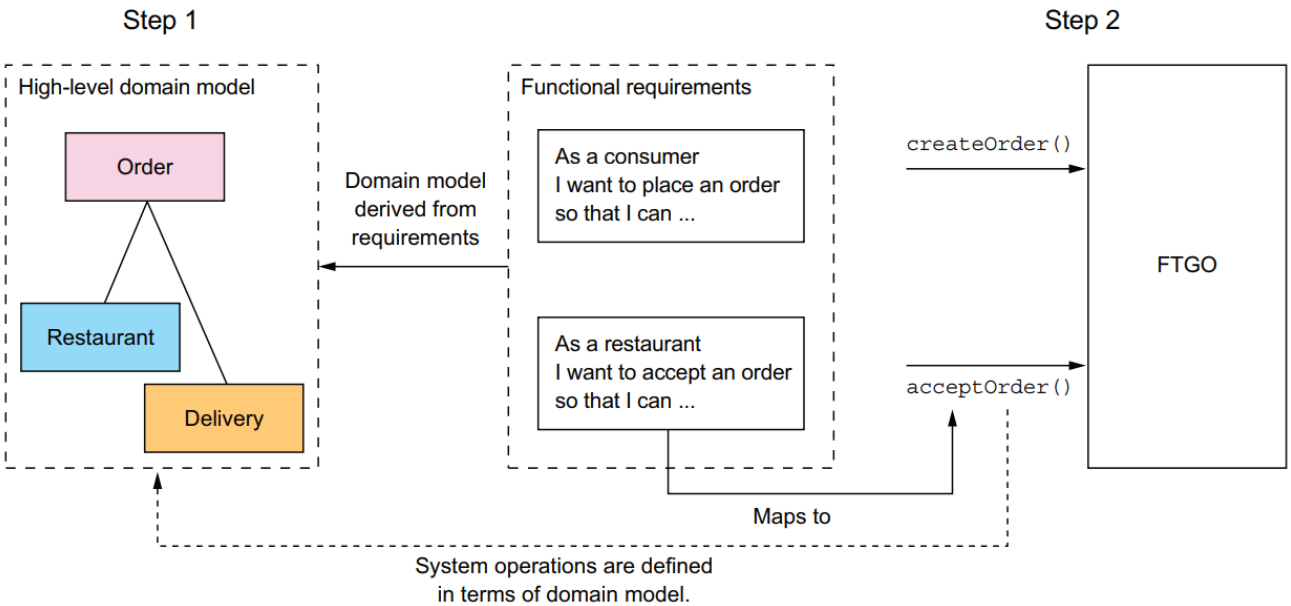

Step 2: Identify services
FTGO
createOrder()
acceptOrder()
Order Service
Restaurant Service
Kitchen Service
...

Step3




There are several obstacles to decomposition. The first is network latency. You might discover that a particular decomposition would be impractical due to too many round-trips between services. Another obstacle to decomposition is that synchronous communication between services reduces availability. You might need to use the concept of self-contained services, described in chapter 3. The third obstacle is the requirement to maintain data consistency across services. You'll typically need to use sagas, discussed in chapter 4. The fourth and final obstacle to decomposition is so-called god classes, which are used throughout an application. Fortunately, you can use concepts from domain-driven design to eliminate god classes.

## Identifying the system operations



the `Place Order` story. We can expand that story into numerous user scenarios including this one:

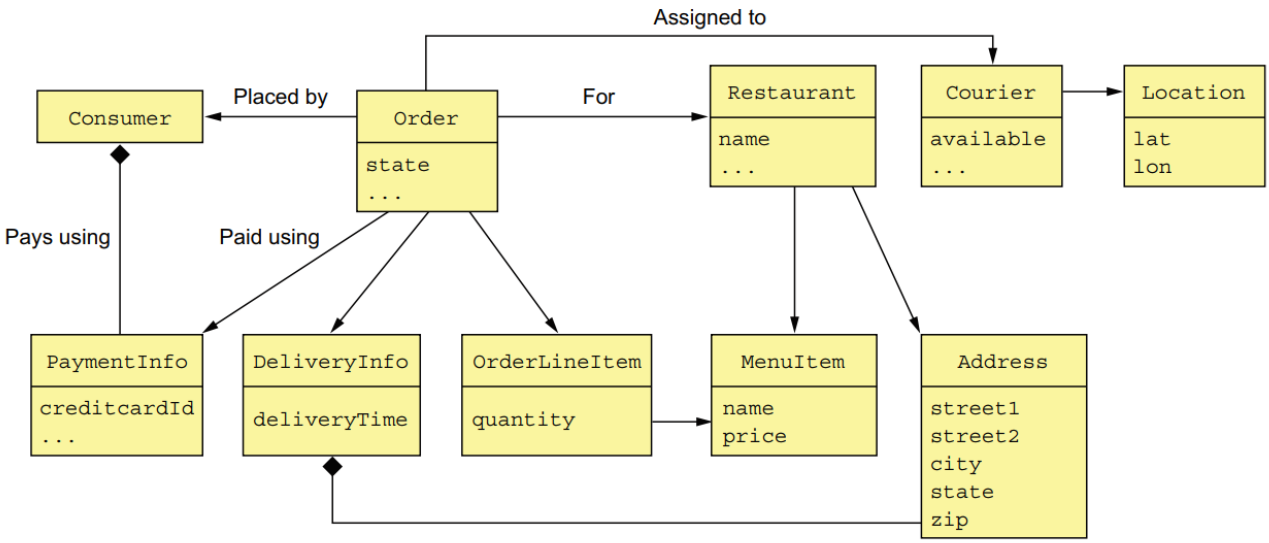
```
Given a consumer
  And a restaurant
  And a delivery address/time that can be served by that restaurant
  And an order total that meets the restaurant's order minimum
When the consumer places an order for the restaurant
Then consumer's credit card is authorized
  And an order is created in the PENDING_ACCEPTANCE state
  And the order is associated with the consumer
  And the order is associated with the restaurant
```

The nouns in this user scenario hint at the existence of various classes, including `Consumer`, `Order`, `Restaurant`, and `CreditCard`.

Similarly, the `Accept Order` story can be expanded into a scenario such as this one:

```
Given an order that is in the PENDING_ACCEPTANCE state
  and a courier that is available to deliver the order
When a restaurant accepts an order with a promise to prepare by a particular
     time
Then the state of the order is changed to ACCEPTED
  And the order's promiseByTime is updated to the promised time
  And the courier is assigned to deliver the order
```

This scenario suggests the existence of `Courier` and `Delivery` classes. The end result after a few iterations of analysis will be a domain model that consists, unsurprisingly, of those classes and others, such as `MenuItem` and `Address`. Figure 2.7 is a class diagram that shows the key classes.




## DEFINING SYSTEM OPERATIONS

There are two types of system operations:

- *Commands*—System operations that create, update, and delete data
- *Queries*—System operations that read (query) data

Ultimately, these system operations will correspond to REST, RPC, or messaging endpoints, but for now thinking of them abstractly is useful. Let's first identify some commands.

A good starting point for identifying system commands is to analyze the verbs in the user stories and scenarios. Consider, for example, the `Place Order` story. It clearly suggests that the system must provide a `Create Order` operation. Many other stories individually map directly to system commands. Table 2.1 lists some of the key system commands.

**Table 2.1 Key system commands for the FTGO application**

| Actor | Story | Command | Description |
|---|---|---|---|
| Consumer | Create Order | `createOrder()` | Creates an order |
| Restaurant | Accept Order | `acceptOrder()` | Indicates that the restaurant has accepted the order and is committed to preparing it by the indicated time |

**Table 2.1 Key system commands for the FTGO application *(continued)***

| Actor | Story | Command | Description |
|---|---|---|---|
| Restaurant | Order Ready for Pickup | `noteOrderReadyForPickup()` | Indicates that the order is ready for pickup |
| Courier | Update Location | `noteUpdatedLocation()` | Updates the current location of the courier |
| Courier | Delivery picked up | `noteDeliveryPickedUp()` | Indicates that the courier has picked up the order |
| Courier | Delivery delivered | `noteDeliveryDelivered()` | Indicates that the courier has delivered the order |

| | |
|---|---|
| Operation | `createOrder` (consumer id, payment method, delivery address, delivery time, restaurant id, order line items) |
| Returns | `orderId`, ... |
| Preconditions | ▪ The consumer exists and can place orders.<br>▪ The line items correspond to the restaurant's menu items.<br>▪ The delivery address and time can be serviced by the restaurant. |
| Post-conditions | ▪ The consumer's credit card was authorized for the order total.<br>▪ An order was created in the `PENDING_ACCEPTANCE` state. |

| | |
|---|---|
| Operation | `acceptOrder(restaurantId, orderId, readyByTime)` |
| Returns | — |
| Preconditions | ▪ The `order.status` is `PENDING_ACCEPTANCE`.<br>▪ A courier is available to deliver the order. |
| Post-conditions | ▪ The `order.status` was changed to `ACCEPTED`.<br>▪ The `order.readyByTime` was changed to the `readyByTime`.<br>▪ The courier was assigned to deliver the order. |

Query:

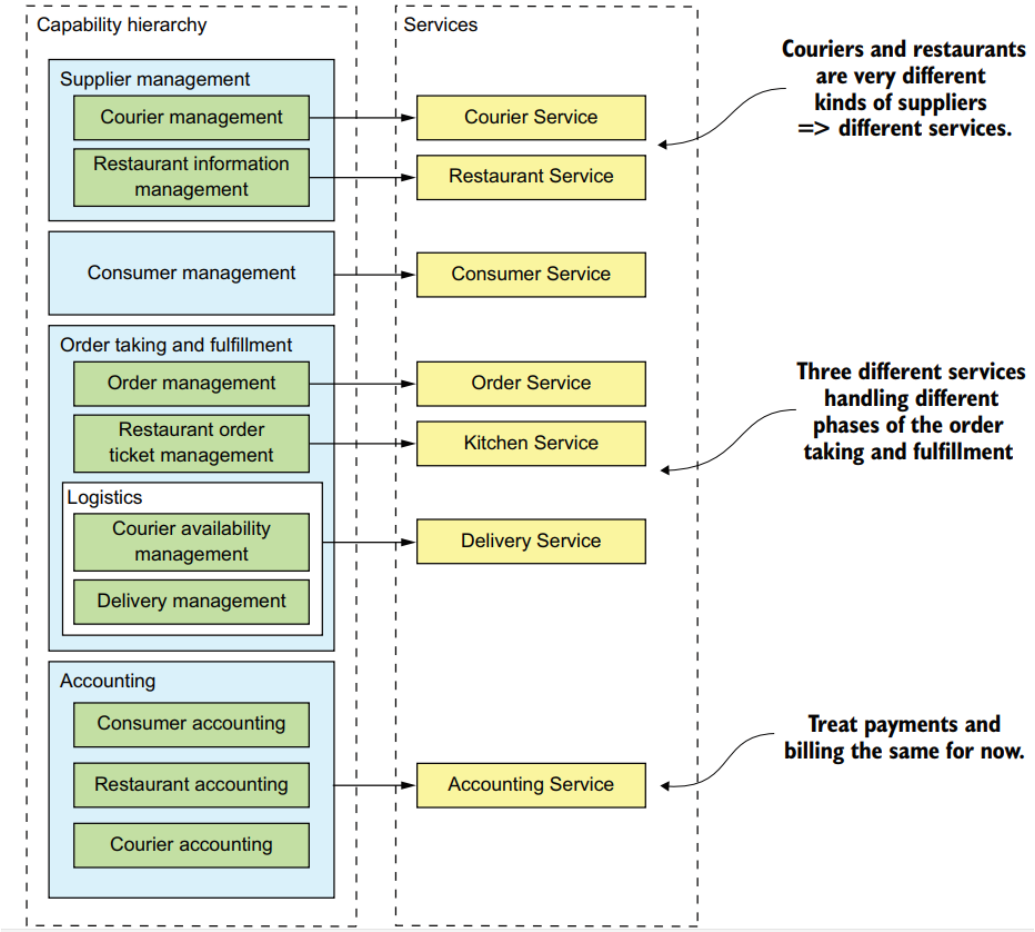
Besides implementing commands, an application must also implement queries. The queries provide the UI with the information a user needs to make decisions. At this stage, we don't have a particular UI design for FTGO application in mind, but consider, for example, the flow when a consumer places an order:

1. User enters delivery address and time.
2. System displays available restaurants.
3. User selects restaurant.
4. System displays menu.
5. User selects item and checks out.
6. System creates order.

This user scenario suggests the following queries:

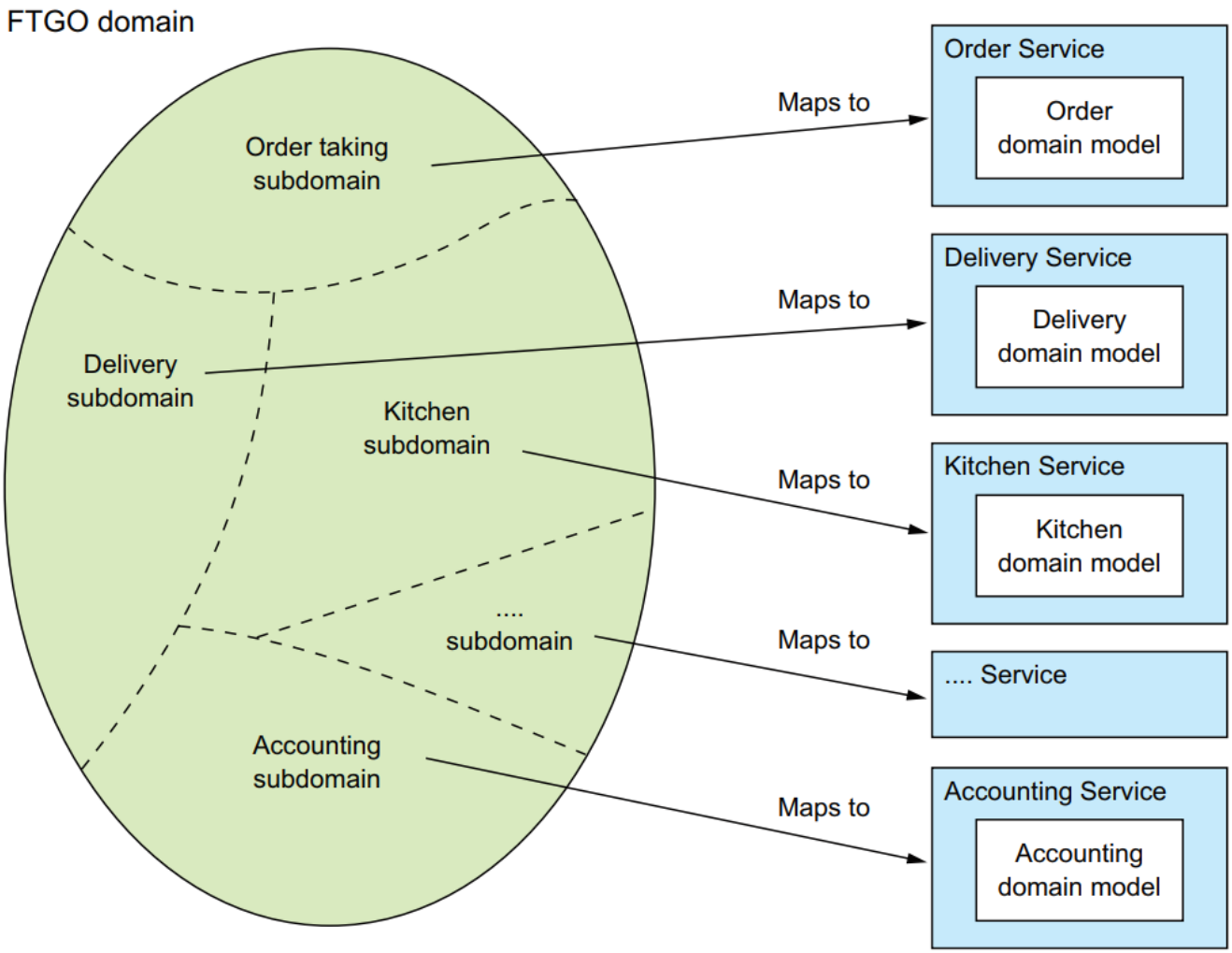
- `findAvailableRestaurants(deliveryAddress, deliveryTime)`—Retrieves the restaurants that can deliver to the specified delivery address at the specified time
- `findRestaurantMenu(id)`—Retrieves information about a restaurant including the menu items

Defining services by applying the Decompose by business capability pattern




Defining services by applying the Decompose by sub-domain pattern

DDD calls the scope of a domain model a *bounded context.* A bounded context includes the code artifacts that implement the model. When using the microservice architecture, each bounded context is a service or possibly a set of services. We can create a microservice architecture by applying DDD and defining a service for each subdomain. Figure 2.9 shows how the subdomains map to services, each with its own domain model.




## Obstacles to decomposing an application into services

On the surface, the strategy of creating a microservice architecture by defining services corresponding to business capabilities or subdomains looks straightforward. You may, however, encounter several obstacles:

- Network latency
- Reduced availability due to synchronous communication
- Maintaining data consistency across services
- Obtaining a consistent view of the data
- God classes preventing decomposition

### Network latency

*Network latency* is an ever-present concern in a distributed system. You might discover that a particular decomposition into services results in a large number of round-trips between two services. Sometimes, you can reduce the latency to an acceptable amount by implementing a batch API for fetching multiple objects in a single round trip. But in other situations, the solution is to combine services, replacing expensive IPC with language-level method or function calls.

### Synchronous interprocess communication reduces availability

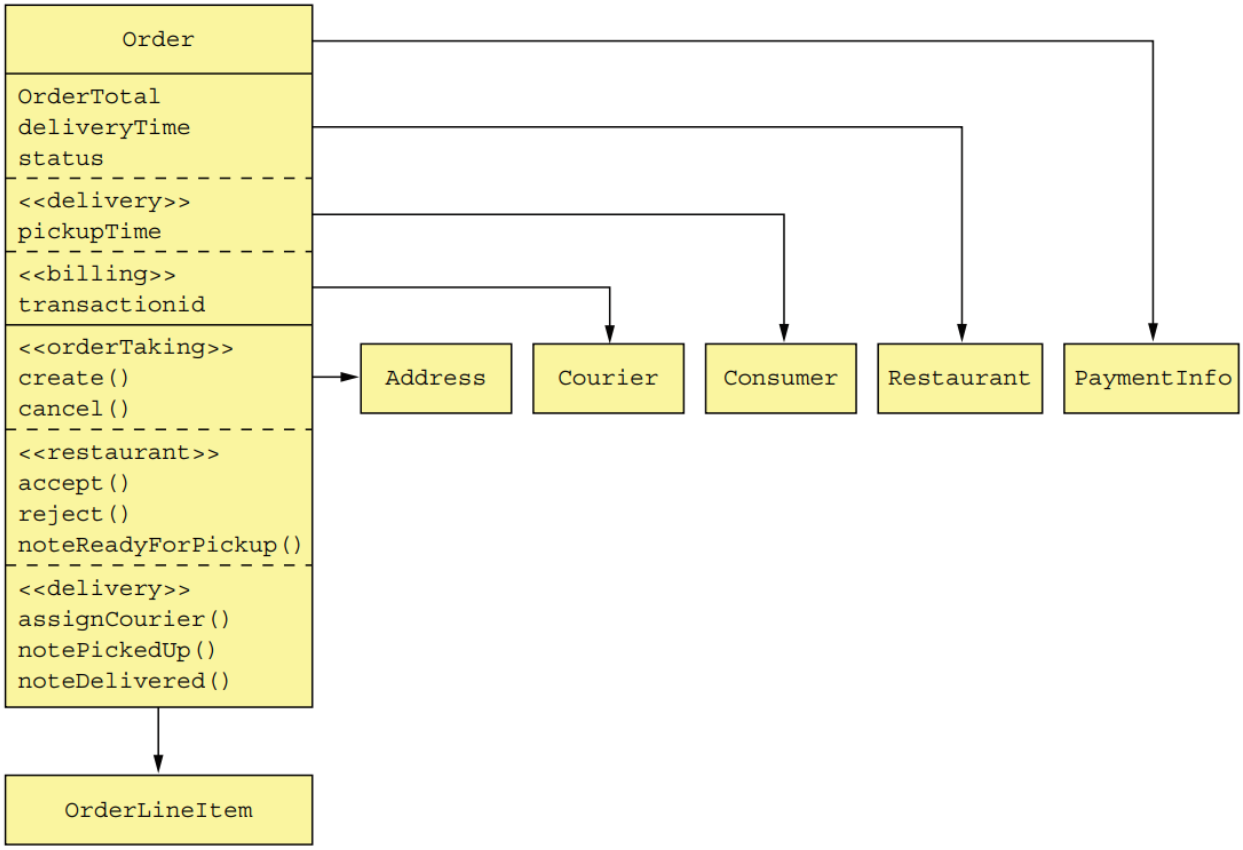
Another problem is how to implement interservice communication in a way that doesn't reduce availability. For example, the most straightforward way to implement the `createOrder()` operation is for the `Order Service` to synchronously invoke the other services using REST. The drawback of using a protocol like REST is that it reduces the availability of the `Order Service`. It won't be able to create an order if any of those other services are unavailable. Sometimes this is a worthwhile trade-off, but in chapter 3 you'll learn that using asynchronous messaging, which eliminates tight coupling and improves availability, is often a better choice.

### Maintaining data consistency across services

Another challenge is maintaining data consistency across services. Some system operations need to update data in multiple services. For example, when a restaurant accepts an order, updates must occur in both the `Kitchen Service` and the `Delivery Service`. The `Kitchen Service` changes the status of the `Ticket`. The `Delivery Service` schedules delivery of the order. Both of these updates must be done atomically.

The traditional solution is to use a two-phase, commit-based, distributed transaction management mechanism. But as you'll see in chapter 4, this is not a good choice for modern applications, and you must use a very different approach to transaction management, a saga. A *saga* is a sequence of local transactions that are coordinated using messaging. Sagas are more complex than traditional ACID transactions but they work well in many situations. One limitation of sagas is that they are eventually consistent. If you need to update some data atomically, then it must reside within a single service, which can be an obstacle to decomposition.

## GOD CLASSES PREVENT DECOMPOSITION




As you can see, the `Order` class has fields and methods corresponding to order processing, restaurant order management, delivery, and payments. This class also has a complex state model, due to the fact that one model has to describe state transitions from disparate parts of the application. In its current form, this class makes it extremely difficult to split code into services.

One solution is to package the `Order` class into a library and create a central `Order` database. All services that process orders use this library and access the access database. The trouble with this approach is that it violates one of the key principles of the microservice architecture and results in undesirable, tight coupling. For example, any change to the `Order` schema requires the teams to update their code in lockstep.

Another solution is to encapsulate the `Order` database in an `Order Service`, which is invoked by the other services to retrieve and update orders. The problem with that design is that the `Order Service` would be a data service with an anemic domain model containing little or no business logic. Neither of these options is appealing, but fortunately, DDD provides a solution.

Using DDD

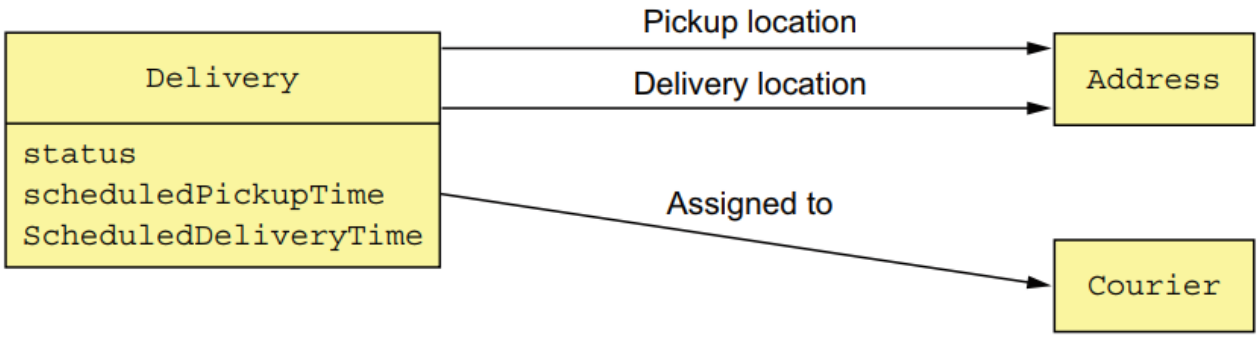



**Figure 2.11 The `Delivery Service` domain model**

Colab between function

| Service | Operations | Collaborators |
|---|---|---|
| `Consumer Service` | `verifyConsumerDetails()` | — |
| `Order Service` | `createOrder()` | ▪ `Consumer Service verifyConsumerDetails()`<br>▪ `Restaurant Service verifyOrderDetails()`<br>▪ `Kitchen Service createTicket()`<br>▪ `Accounting Service authorizeCard()` |
| `Restaurant Service` | ▪ `findAvailableRestaurants()`<br>▪ `verifyOrderDetails()` | — |
| `Kitchen Service` | ▪ `createTicket()`<br>▪ `acceptOrder()`<br>▪ `noteOrderReadyForPickup()` | ▪ `Delivery Service scheduleDelivery()` |
| `Delivery Service` | ▪ `scheduleDelivery()`<br>▪ `noteUpdatedLocation()`<br>▪ `noteDeliveryPickedUp()`<br>▪ `noteDeliveryDelivered()` | — |
| `Accounting Service` | ▪ `authorizeCard()` | — |

## 2.3 Interprocess communication in microservices architect

### 2.3.1 Overview of interprocess communication in a microservice architecture

There are lots of different IPC technologies to choose from. Services can use synchronous request/response-based communication mechanisms, such as HTTP-based REST or gRPC. Alternatively, they can use asynchronous, message-based communication mechanisms such as AMQP or STOMP. There are also a variety of different messages formats. Services can use human-readable, text-based formats such as JSON or XML. Alternatively, they could use a more efficient binary format such as Avro or Protocol Buffers.

#### Interaction styles

There are a variety of client-service interaction styles. As table 3.1 shows, they can be categorized in two dimensions. The first dimension is whether the interaction is one-to-one or one-to-many:

- *One-to-one*—Each client request is processed by exactly one service.
- *One-to-many*—Each request is processed by multiple services.

The second dimension is whether the interaction is synchronous or asynchronous:

- *Synchronous*—The client expects a timely response from the service and might even block while it waits.
- *Asynchronous*—The client doesn't block, and the response, if any, isn't necessarily sent immediately.

**Table 3.1 The various interaction styles can be characterized in two dimensions: one-to-one vs one-to-many and synchronous vs asynchronous.**

| | one-to-one | one-to-many |
|---|---|---|
| Synchronous | Request/response | — |
| Asynchronous | Asynchronous request/response<br>One-way notifications | Publish/subscribe<br>Publish/async responses |

### 2.3.2 Communication via sync message

#### Message formats

The essence of IPC is the exchange of messages. *Messages* usually contain data, and so an important design decision is the format of that data. The choice of message format can impact the efficiency of IPC, the usability of the API, and its evolvability. If you're using a messaging system or protocols such as HTTP, you get to pick your message format. Some IPC mechanisms—such as gRPC, which you'll learn about shortly—might dictate the message format. In either case, it's essential to use a cross-language message format. Even if you're writing your microservices in a single language today, it's likely that you'll use other languages in the future. You shouldn't, for example, use Java serialization.

- TEXT-BASED MESSAGE FORMATS

  The structure of XML documents is specified by an XML schema (www.w3.org/XML/Schema). Over time, the developer community has come to realize that JSON also needs a similar mechanism. One popular option is to use the JSON Schema standard (http://json-schema.org). A JSON schema defines the names and types of a message's properties and whether they're optional or required. As well as being useful documentation, a JSON schema can be used by an application to validate incoming messages.

- BINARY MESSAGE FORMATS

There are several different binary formats to choose from. Popular formats include Protocol Buffers (https://developers.google.com/protocol-buffers/docs/overview) and Avro (https://avro.apache.org). Both formats provide a typed IDL for defining the structure of your messages. A compiler then generates the code that serializes and deserializes the messages. You're forced to take an API-first approach to service design! Moreover, if you write your client in a statically typed language, the compiler checks that it uses the API correctly.

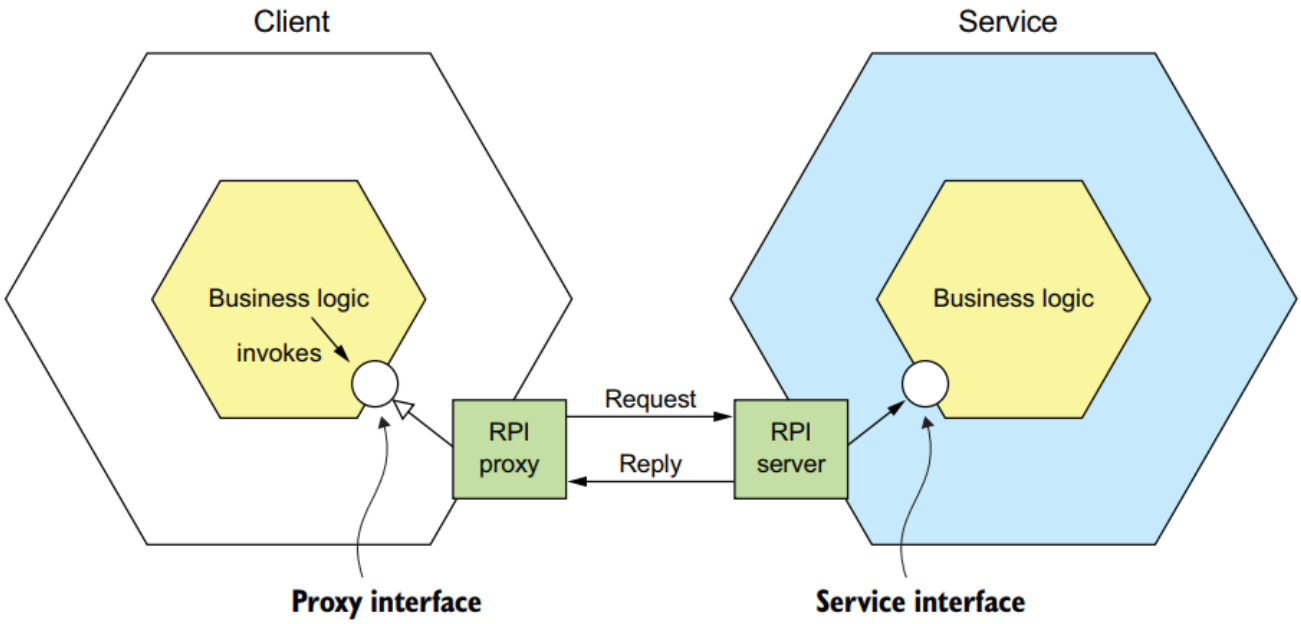
One difference between these two binary formats is that Protocol Buffers uses tagged fields, whereas an Avro consumer needs to know the schema in order to interpret messages. As a result, handling API evolution is easier with Protocol Buffers than with Avro. This blog post (http://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html) is an excellent comparison of Thrift, Protocol Buffers, and Avro.

**Communicating using the synchronous Remote procedure invocation pattern**

**Pattern: Remote procedure invocation**

A client invokes a service using a synchronous, remote procedure invocation-based protocol, such as REST (http://microservices.io/patterns/communication-style/messaging.html).

Figure 3.1 shows how RPI works. The business logic in the client invokes a *proxy interface*, implemented by an *RPI proxy* adapter class. The *RPI proxy* makes a request to the service. The request is handled by an *RPI server* adapter class, which invokes the service's business logic via an interface. It then sends back a reply to the *RPI proxy*, which returns the result to the client's business logic.




Using REST

### THE REST MATURITY MODEL

Leonard Richardson (no relation to your author) defines a very useful maturity model for REST (http://martinfowler.com/articles/richardsonMaturityModel.html) that consists of the following levels:

- *Level 0*—Clients of a level 0 service invoke the service by making HTTP POST requests to its sole URL endpoint. Each request specifies the action to perform, the target of the action (for example, the business object), and any parameters.
- *Level 1*—A level 1 service supports the idea of resources. To perform an action on a resource, a client makes a POST request that specifies the action to perform and any parameters.
- *Level 2*—A level 2 service uses HTTP verbs to perform actions: GET to retrieve, POST to create, and PUT to update. The request query parameters and body, if any, specify the actions' parameters. This enables services to use web infrastructure such as caching for GET requests.
- *Level 3*—The design of a level 3 service is based on the terribly named HATEOAS (Hypertext As The Engine Of Application State) principle. The basic idea is that the representation of a resource returned by a GET request contains links for performing actions on that resource. For example, a client can cancel an order using a link in the representation returned by the GET request that retrieved the order. The benefits of HATEOAS include no longer having to hard-wire URLs into client code (www.infoq.com/news/2009/04/hateoas-restful-api-advantages).

### SPECIFYING REST APIS

As mentioned earlier in section 3.1, you must define your APIs using an interface definition language (IDL). Unlike older communication protocols like CORBA and SOAP, REST did not originally have an IDL. Fortunately, the developer community has rediscovered the value of an IDL for RESTful APIs. The most popular REST IDL is the Open API Specification (www.openapis.org), which evolved from the Swagger open source project. The Swagger project is a set of tools for developing and documenting REST APIs. It includes tools that generate client stubs and server skeletons from an interface definition.

## THE CHALLENGE OF FETCHING MULTIPLE RESOURCES IN A SINGLE REQUEST

REST resources are usually oriented around business objects, such as `Consumer` and `Order`. Consequently, a common problem when designing a REST API is how to

enable the client to retrieve multiple related objects in a single request. For example, imagine that a REST client wanted to retrieve an `Order` and the `Order`'s `Consumer`. A pure REST API would require the client to make at least two requests, one for the `Order` and another for its `Consumer`. A more complex scenario would require even more round-trips and suffer from excessive latency.

One solution to this problem is for an API to allow the client to retrieve related resources when it gets a resource. For example, a client could retrieve an `Order` and its `Consumer` using `GET /orders/order-id-1345?expand=consumer`. The query parameter specifies the related resources to return with the `Order`. This approach works well in many scenarios but it's often insufficient for more complex scenarios. It's also potentially time consuming to implement. This has led to the increasing popularity of alternative API technologies such as GraphQL (http://graphql.org) and Netflix Falcor (http://netflix.github.io/falcor/), which are designed to support efficient data fetching.

#### THE CHALLENGE OF MAPPING OPERATIONS TO HTTP VERBS

Another common REST API design problem is how to map the operations you want to perform on a business object to an HTTP verb. A REST API should use PUT for updates, but there may be multiple ways to update an order, including cancelling it, revising the order, and so on. Also, an update might not be idempotent, which is a requirement for using PUT. One solution is to define a sub-resource for updating a particular aspect of a resource. The `Order Service`, for example, has a `POST /orders/{orderId}/cancel` endpoint for cancelling orders, and a `POST /orders/{orderId}/revise` endpoint for revising orders. Another solution is to specify a verb as a URL query parameter. Sadly, neither solution is particularly RESTful.

### Using gRPC

As mentioned in the preceding section, one challenge with using REST is that because HTTP only provides a limited number of verbs, it's not always straightforward to design a REST API that supports multiple update operations. An IPC technology that avoids this issue is gRPC (www.grpc.io), a framework for writing cross-language clients and servers (see https://en.wikipedia.org/wiki/Remote_procedure_call for more). gRPC is a binary message-based protocol, and this means—as mentioned earlier in the discussion of binary message formats—you're forced to take an API-first approach to service design. You define your gRPC APIs using a Protocol Buffers-based IDL, which is Google's language-neutral mechanism for serializing structured data. You use the Protocol Buffer compiler to generate client-side stubs and server-side skeletons. The compiler can generate code for a variety of languages, including Java, C#, NodeJS, and GoLang. Clients and servers exchange binary messages in the Protocol Buffers format using HTTP/2.

gRPC has several benefits:

- It's straightforward to design an API that has a rich set of update operations.
- It has an efficient, compact IPC mechanism, especially when exchanging large messages.
- Bidirectional streaming enables both RPI and messaging styles of communication.
- It enables interoperability between clients and services written in a wide range of languages.

gRPC also has several drawbacks:

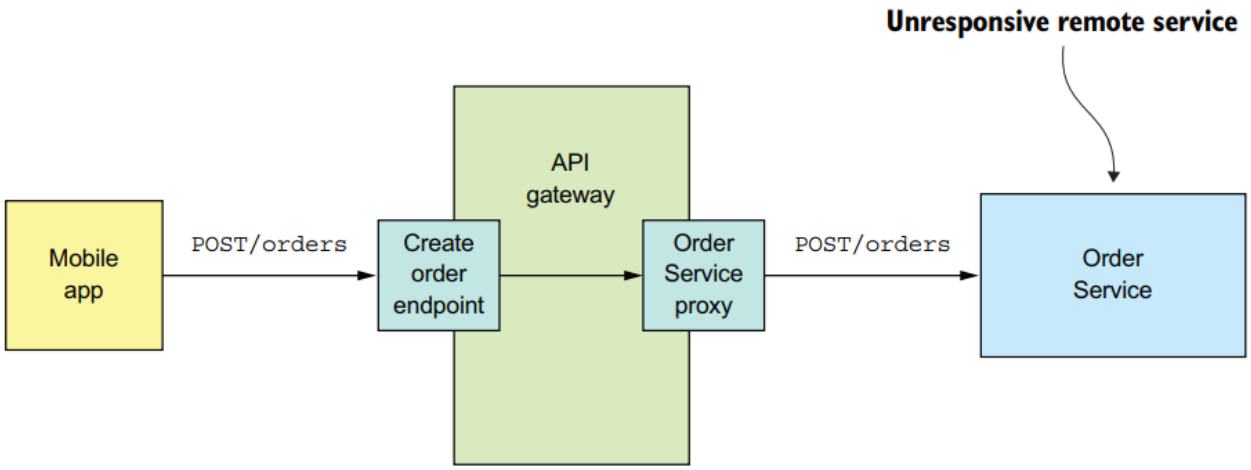
- It takes more work for JavaScript clients to consume gRPC-based API than REST/JSON-based APIs.
- Older firewalls might not support HTTP/2.

gRPC is a compelling alternative to REST, but like REST, it's a synchronous communication mechanism, so it also suffers from the problem of partial failure. Let's take a look at what that is and how to handle it.

## Handling partial failure using the Circuit breaker pattern

**Pattern: Circuit breaker**

An RPI proxy that immediately rejects invocations for a timeout period after the number of consecutive failures exceeds a specified threshold. See http://microservices.io/patterns/reliability/circuit-breaker.html.



A naive implementation of the `OrderServiceProxy` would block indefinitely, waiting for a response. Not only would that result in a poor user experience, but in many applications it would consume a precious resource, such as a thread. Eventually the API gateway would run out of resources and become unable to handle requests. The entire API would be unavailable.

It's essential that you design your services to prevent partial failures from cascading throughout the application. There are two parts to the solution:

- You must use design RPI proxies, such as `OrderServiceProxy`, to handle unresponsive remote services.
- You need to decide how to recover from a failed remote service.

#### DEVELOPING ROBUST RPI PROXIES

Whenever one service synchronously invokes another service, it should protect itself using the approach described by Netflix (http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html). This approach consists of a combination of the following mechanisms:

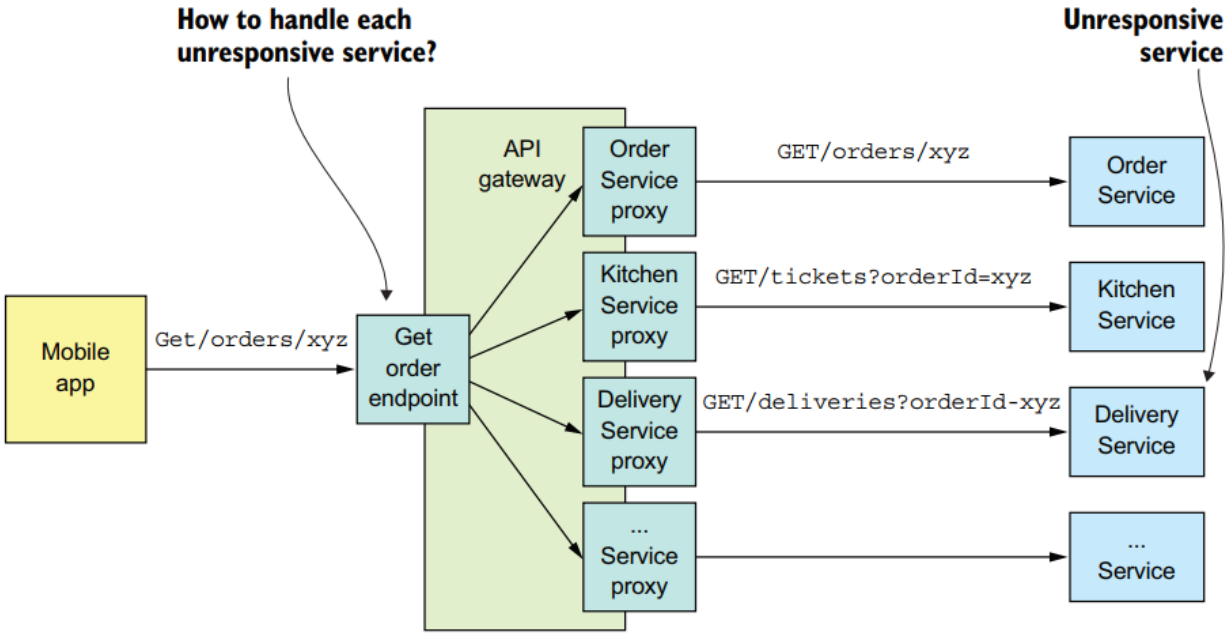
- *Network timeouts*—Never block indefinitely and always use timeouts when waiting for a response. Using timeouts ensures that resources are never tied up indefinitely.
- *Limiting the number of outstanding requests from a client to a service*—Impose an upper bound on the number of outstanding requests that a client can make to a particular service. If the limit has been reached, it's probably pointless to make additional requests, and those attempts should fail immediately.
- *Circuit breaker pattern*—Track the number of successful and failed requests, and if the error rate exceeds some threshold, trip the circuit breaker so that further attempts fail immediately. A large number of requests failing suggests that the service is unavailable and that sending more requests is pointless. After a timeout period, the client should try again, and, if successful, close the circuit breaker.

Netflix Hystrix (https://github.com/Netflix/Hystrix) is an open source library that implements these and other patterns. If you're using the JVM, you should definitely consider using Hystrix when implementing RPI proxies. And if you're running in a non-JVM environment, you should use an equivalent library. For example, the Polly library is popular in the .NET community (https://github.com/App-vNext/Polly).

### RECOVERING FROM AN UNAVAILABLE SERVICE

Using a library such as Hystrix is only part of the solution. You must also decide on a case-by-case basis how your services should recover from an unresponsive remote service. One option is for a service to simply return an error to its client. For example, this approach makes sense for the scenario shown in figure 3.2, where the request to create an `Order` fails. The only option is for the API gateway to return an error to the mobile client.

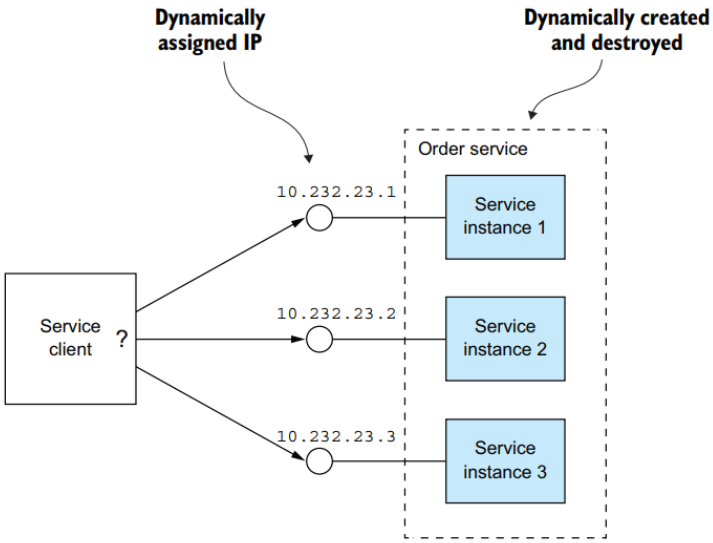
In other scenarios, returning a fallback value, such as either a default value or a cached response, may make sense. For example, chapter 7 describes how the API gateway could implement the `findOrder()` query operation by using the API composition pattern. As figure 3.3 shows, its implementation of the `GET /orders/{orderId}` endpoint invokes several services, including the `Order Service`, `Kitchen Service`, and `Delivery Service`, and combines the results.




## Using service discovery

Say you're writing some code that invokes a service that has a REST API. In order to make a request, your code needs to know the network location (IP address and port) of a service instance. In a traditional application running on physical hardware, the network locations of service instances are usually static. For example, your code could read the network locations from a configuration file that's occasionally updated. But in a modern, cloud-based microservices application, it's usually not that simple. As is shown in figure 3.4, a modern application is much more dynamic.

Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of autoscaling, failures, and upgrades. Consequently, your client code must use a service discovery.




### OVERVIEW OF SERVICE DISCOVERY

As you've just seen, you can't statically configure a client with the IP addresses of the services. Instead, an application must use a dynamic service discovery mechanism. Service discovery is conceptually quite simple: its key component is a service registry, which is a database of the network locations of an application's service instances.

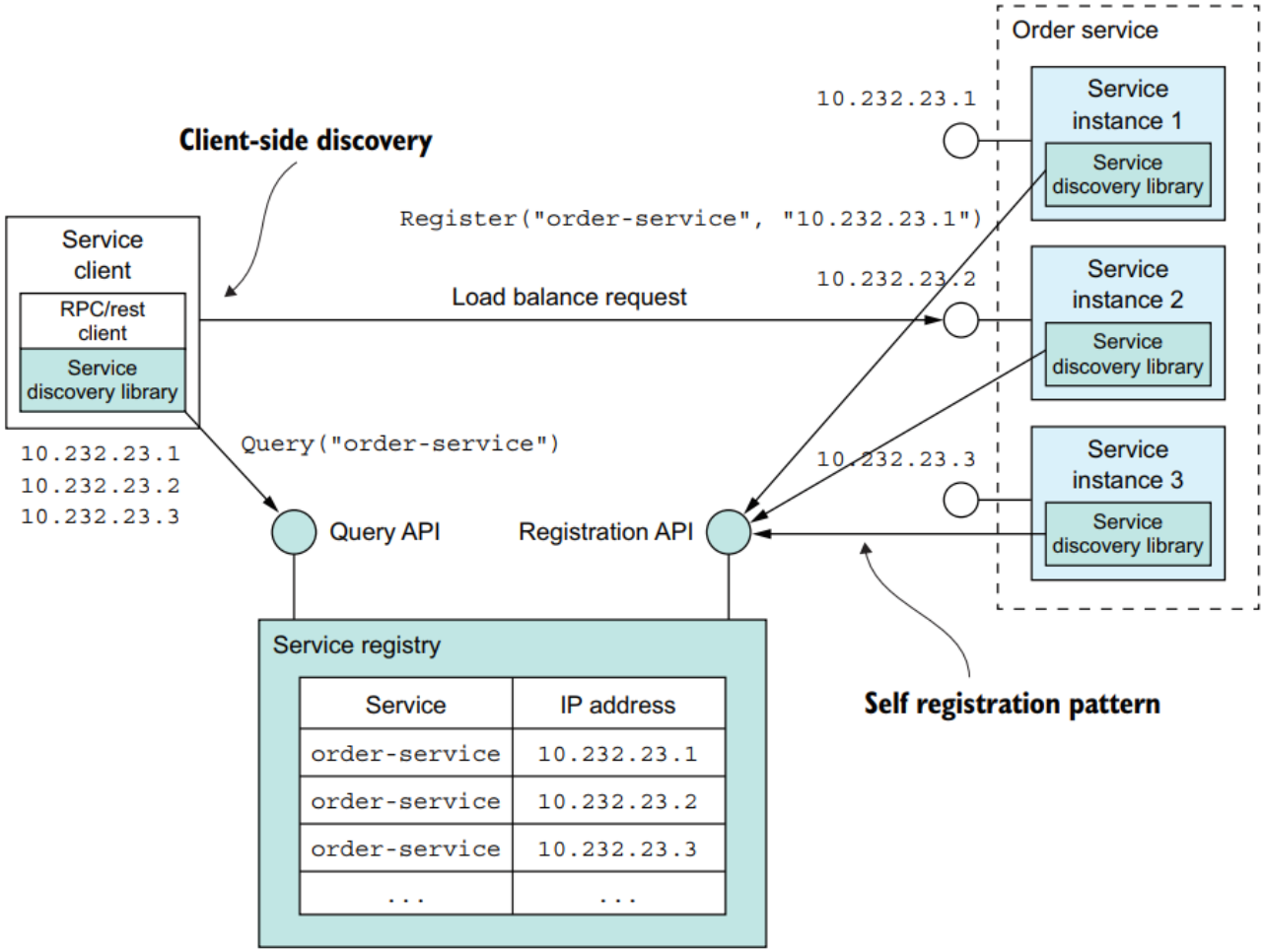
The service discovery mechanism updates the service registry when service instances start and stop. When a client invokes a service, the service discovery mechanism queries the service registry to obtain a list of available service instances and routes the request to one of them.

There are two main ways to implement service discovery:

- The services and their clients interact directly with the service registry.
- The deployment infrastructure handles service discovery. (I talk more about that in chapter 12.)

### APPLYING THE APPLICATION-LEVEL SERVICE DISCOVERY PATTERNS

One way to implement service discovery is for the application's services and their clients to interact with the service registry. Figure 3.5 shows how this works. A service instance registers its network location with the service registry. A service client invokes a service by first querying the service registry to obtain a list of service instances. It then sends a request to one of those instances.




Self registration pattern

This approach to service discovery is a combination of two patterns. The first pattern is the Self registration pattern. A service instance invokes the service registry's registration API to register its network location. It may also supply a health check URL, described in more detail in chapter 11. The *health check* URL is an API endpoint that the service registry invokes periodically to verify that the service instance is healthy and available to handle requests. A service registry may require a service instance to periodically invoke a "heartbeat" API in order to prevent its registration from expiring.

**Pattern: Self registration**

A service instance registers itself with the service registry. See http://microservices.io/patterns/self-registration.html.

Client-side discovery pattern

The second pattern is the Client-side discovery pattern. When a service client wants to invoke a service, it queries the service registry to obtain a list of the service's instances. To improve performance, a client might cache the service instances. The service client

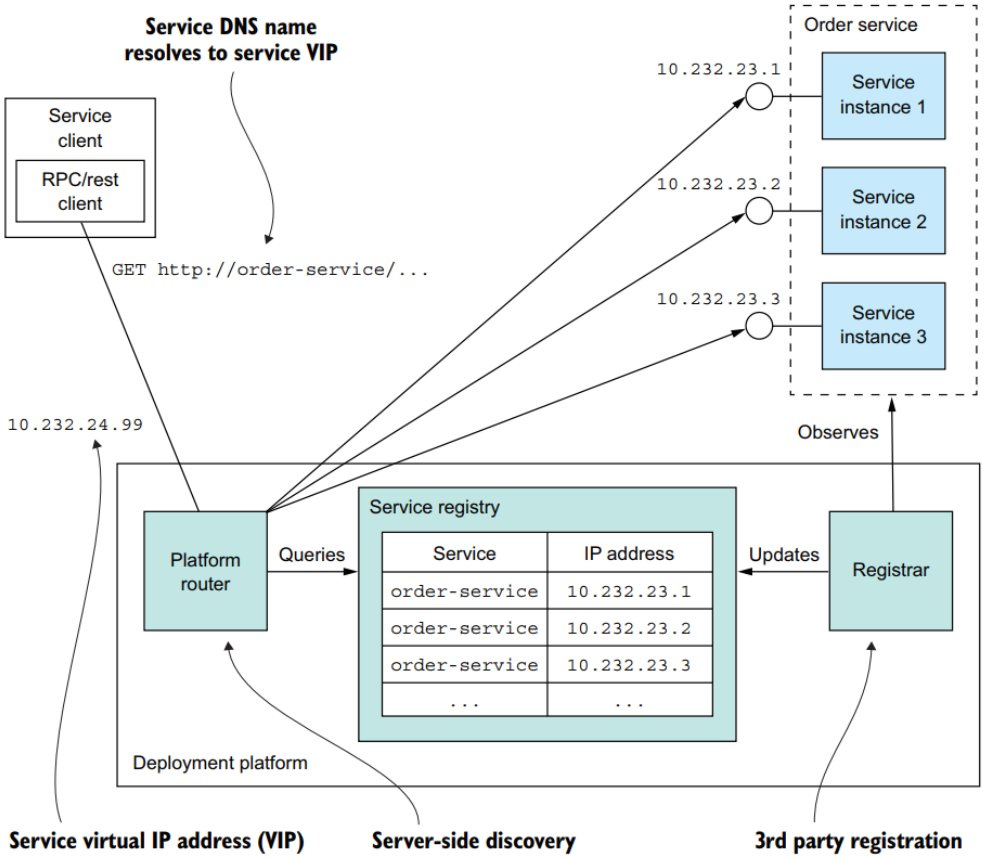
then uses a load-balancing algorithm, such as a round-robin or random, to select a service instance. It then makes a request to a select service instance.

**Pattern: Client-side discovery**

A service client retrieves the list of available service instances from the service registry and load balances across them. See http://microservices.io/patterns/client-side-discovery.html.

## APPLYING THE PLATFORM-PROVIDED SERVICE DISCOVERY PATTERNS

Later in chapter 12 you'll learn that many modern deployment platforms such as Docker and Kubernetes have a built-in service registry and service discovery mechanism. The deployment platform gives each service a DNS name, a virtual IP (VIP) address, and a DNS name that resolves to the VIP address. A service client makes a request to the DNS name/VIP, and the deployment platform automatically routes the request to one of the available service instances. As a result, service registration, service discovery, and request routing are entirely handled by the deployment platform.




This approach is:

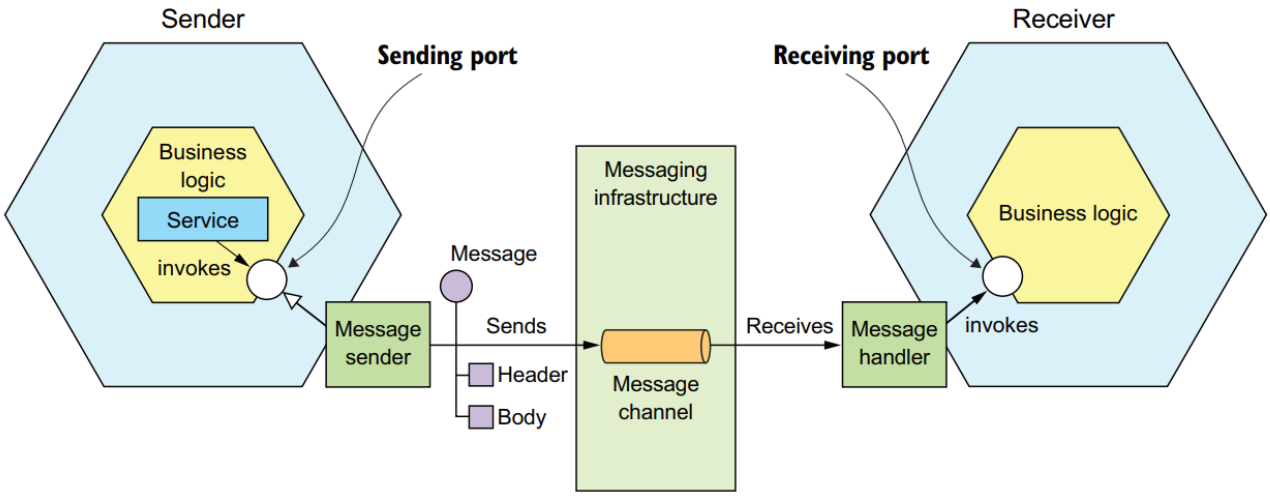
- Server side service discovery
- 3rd party registration pattern

Communicating using the Asynchronous messaging pattern

### 2.3.3 Communication via async message

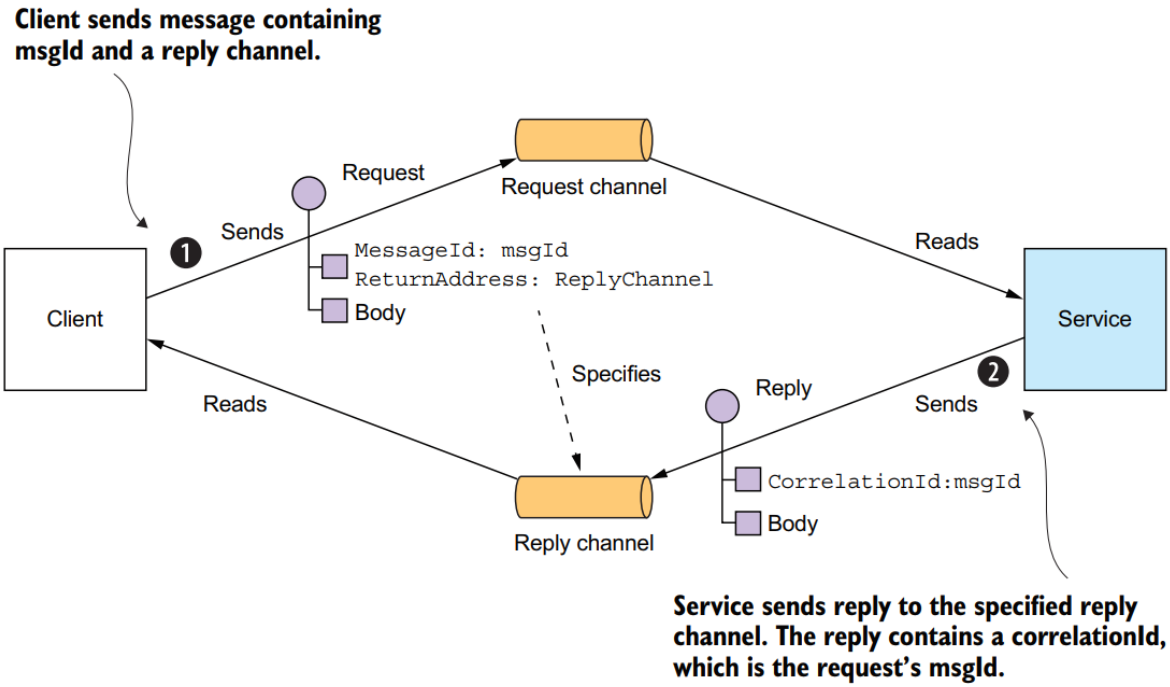
When using messaging, services communicate by asynchronously exchanging messages. A messaging-based application typically uses a *message broker*, which acts as an intermediary between the services, although another option is to use a brokerless architecture, where the services communicate directly with each other. A service client makes a request to a service by sending it a message. If the service instance is expected to reply, it will do so by sending a separate message back to the client. Because the communication is asynchronous, the client doesn't block waiting for a reply. Instead, the client is written assuming that the reply won't be received immediately.

Overview of messaging




There are two kinds of channels: point-to-point (www.enterpriseintegrationpatterns.com/PointToPointChannel.html) and publish-subscribe (www.enterpriseintegration-patterns.com/PublishSubscribeChannel.html):

- A *point-to-point* channel delivers a message to exactly one of the consumers that is reading from the channel. Services use point-to-point channels for the one-to-one interaction styles described earlier. For example, a command message is often sent over a point-to-point channel.
- A *publish-subscribe* channel delivers each message to all of the attached consumers. Services use publish-subscribe channels for the one-to-many interaction styles described earlier. For example, an event message is usually sent over a publish-subscribe channel.



### Implementing one-way notifications

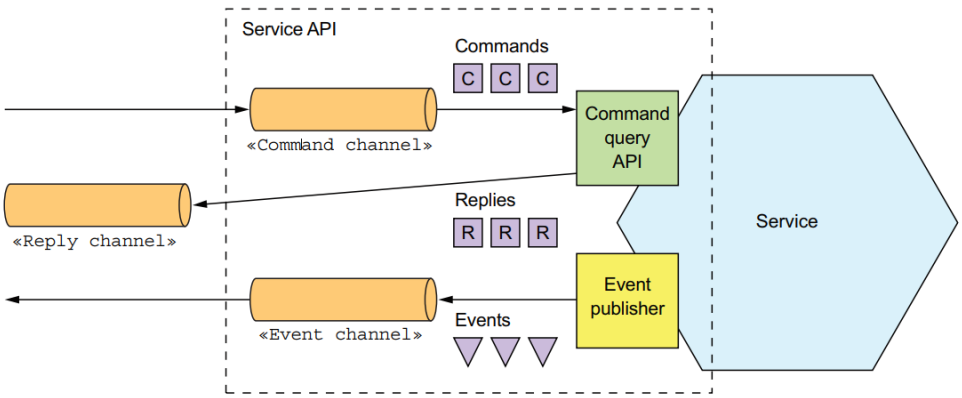
Implementing one-way notifications is straightforward using asynchronous messaging. The client sends a message, typically a command message, to a point-to-point channel owned by the service. The service subscribes to the channel and processes the message. It doesn't send back a reply.

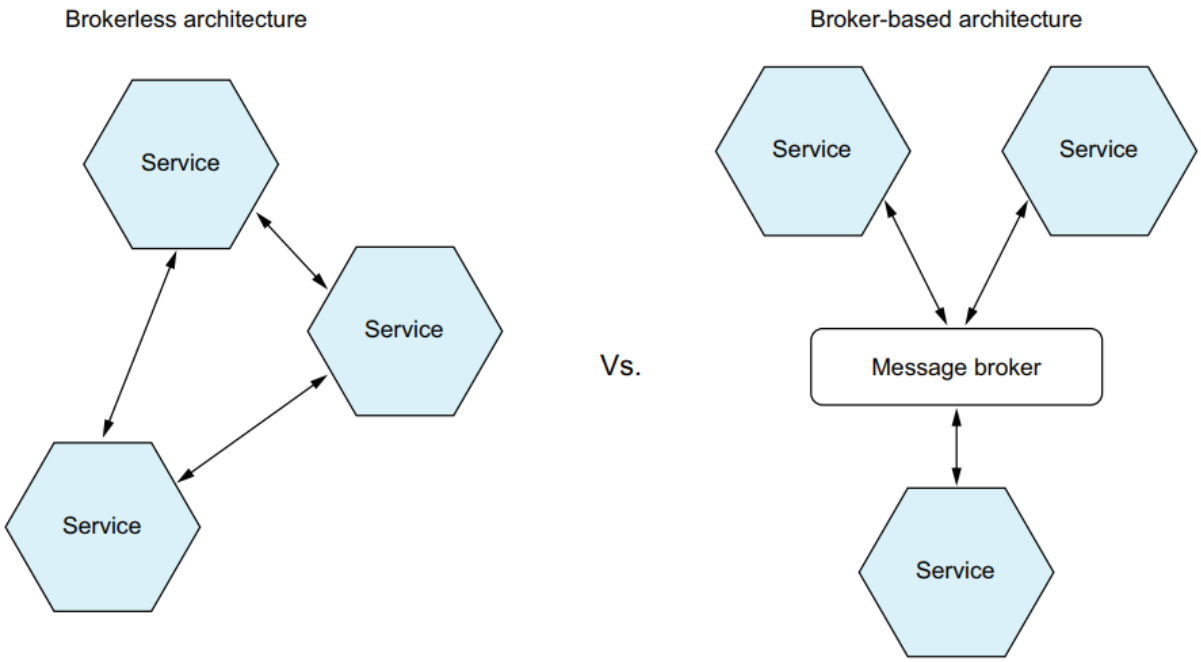
### Implementing publish/subscribe

Messaging has built-in support for the publish/subscribe style of interaction. A client publishes a message to a publish-subscribe channel that is read by multiple consumers. As described in chapters 4 and 5, services use publish/subscribe to publish domain events, which represent changes to domain objects. The service that publishes the domain events owns a publish-subscribe channel, whose name is derived from the domain class. For example, the `Order Service` publishes `Order` events to an `Order` channel, and the `Delivery Service` publishes `Delivery` events to a `Delivery` channel. A service that's interested in a particular domain object's events only has to subscribe to the appropriate channel.

### Implementing publish/async responses

The publish/async responses interaction style is a higher-level style of interaction that's implemented by combining elements of publish/subscribe and request/response. A client publishes a message that specifies a *reply channel* header to a publish-subscribe channel. A consumer writes a reply message containing a *correlation id* to the reply channel. The client gathers the responses by using the *correlation id* to match the reply messages with the request.






## OVERVIEW OF BROKER-BASED MESSAGING

There are many message brokers to chose from. Examples of popular open source message brokers include the following:

- ActiveMQ (http://activemq.apache.org)
- RabbitMQ (https://www.rabbitmq.com)
- Apache Kafka (http://kafka.apache.org)

There are also cloud-based messaging services, such as AWS Kinesis (https://aws.amazon.com/kinesis/) and AWS SQS (https://aws.amazon.com/sqs/).

When selecting a message broker, you have various factors to consider, including the following:

- *Supported programming languages*—You probably should pick one that supports a variety of programming languages.
- *Supported messaging standards*—Does the message broker support any standards, such as AMQP and STOMP, or is it proprietary?
- *Messaging ordering*—Does the message broker preserve ordering of messages?
- *Delivery guarantees*—What kind of delivery guarantees does the broker make?
- *Persistence*—Are messages persisted to disk and able to survive broker crashes?
- *Durability*—If a consumer reconnects to the message broker, will it receive the messages that were sent while it was disconnected?
- *Scalability*—How scalable is the message broker?
- *Latency*—What is the end-to-end latency?
- *Competing consumers*—Does the message broker support competing consumers?

#### IMPLEMENTING MESSAGE CHANNELS USING A MESSAGE BROKER

Each message broker implements the message channel concept in a different way. As table 3.2 shows, JMS message brokers such as ActiveMQ have queues and topics. AMQP-based message brokers such as RabbitMQ have exchanges and queues. Apache Kafka has topics, AWS Kinesis has streams, and AWS SQS has queues. What's more, some message brokers offer more flexible messaging than the message and channels abstraction described in this chapter.

**Table 3.2 Each message broker implements the message channel concept in a different way.**

| Message broker | Point-to-point channel | Publish-subscribe channel |
|---|---|---|
| JMS | Queue | Topic |
| Apache Kafka | Topic | Topic |
| AMQP-based brokers, such as RabbitMQ | Exchange + Queue | Fanout exchange and a queue per consumer |
| AWS Kinesis | Stream | Stream |
| AWS SQS | Queue | — |

Competing receivers and message ordering

One challenge is how to scale out message receivers while preserving message ordering. It's a common requirement to have multiple instances of a service in order to process messages concurrently. Moreover, even a single service instance will probably use threads to concurrently process multiple messages. Using multiple threads and service instances to concurrently process messages increases the throughput of the application. But the challenge with processing messages concurrently is ensuring that each message is processed once and in order.

For example, imagine that there are three instances of a service reading from the same point-to-point channel and that a sender publishes `Order Created`, `Order Updated`, and `Order Cancelled` event messages sequentially. A simplistic messaging implementation could concurrently deliver each message to a different receiver. Because of delays due to network issues or garbage collections, messages might be processed out of order, which would result in strange behavior. In theory, a service instance might process the `Order Cancelled` message before another service processes the `Order Created` message!

A common solution, used by modern message brokers like Apache Kafka and AWS Kinesis, is to use *sharded* (partitioned) channels. Figure 3.11 shows how this works. There are three parts to the solution:

1. A sharded channel consists of two or more shards, each of which behaves like a channel.
2. The sender specifies a shard key in the message's header, which is typically an arbitrary string or sequence of bytes. The message broker uses a shard key to assign the message to a particular shard/partition. It might, for example, select the shard by computing the hash of the shard key modulo the number of shards.
3. The messaging broker groups together multiple instances of a receiver and treats them as the same logical receiver. Apache Kafka, for example, uses the term *consumer group*. The message broker assigns each shard to a single receiver. It reassigns shards when receivers start up and shut down.

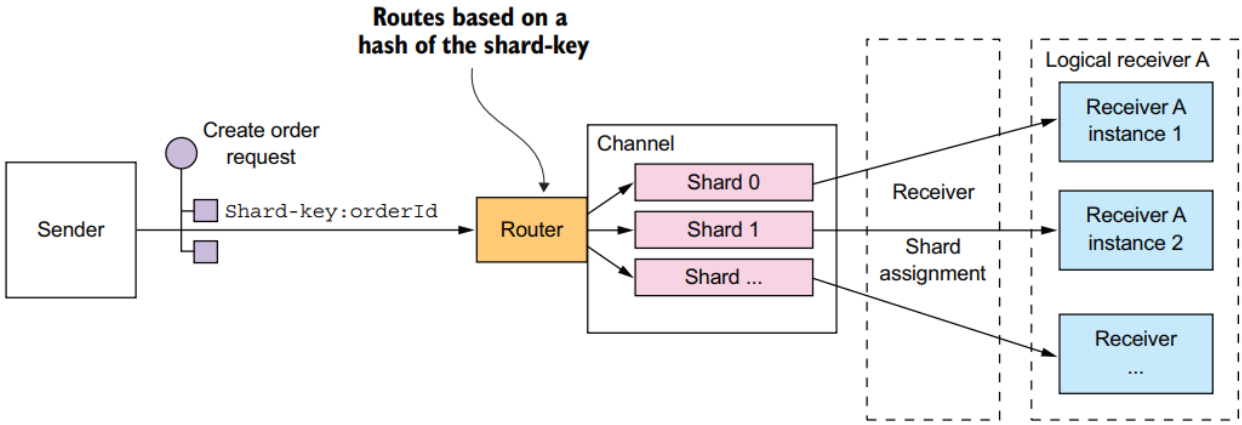



**Figure 3.11 Scaling consumers while preserving message ordering by using a sharded (partitioned) message channel. The sender includes the shard key in the message. The message broker writes the message to a shard determined by the shard key. The message broker assigns each partition to an instance of the replicated receiver.**

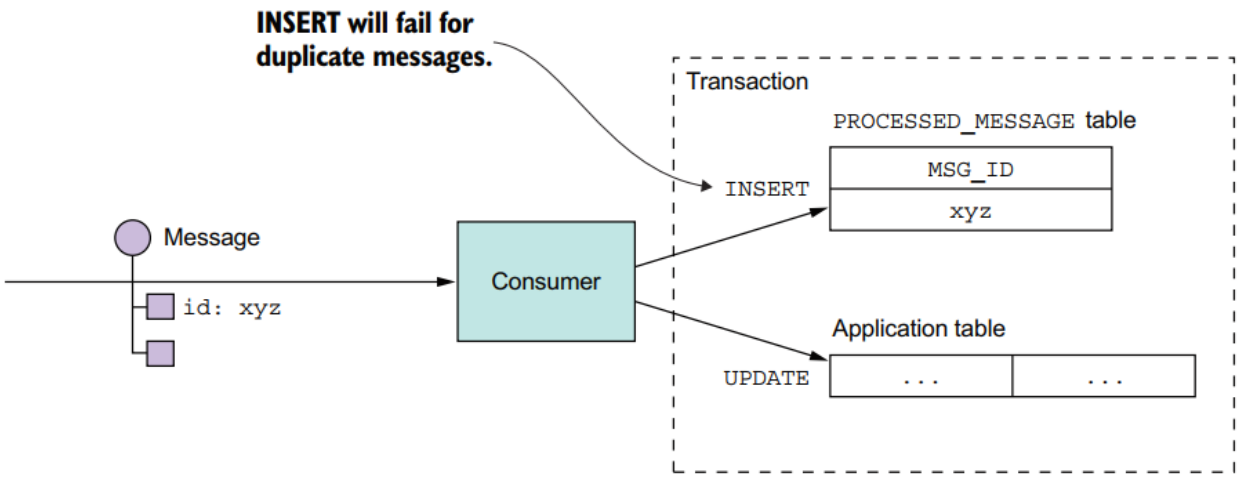
In this example, each `Order` event message has the `orderId` as its shard key. Each event for a particular order is published to the same shard, which is read by a single consumer instance. As a result, these messages are guaranteed to be processed in order.

## Handling duplicate messages

There are a couple of different ways to handle duplicate messages:

- Write idempotent message handlers.
- Track messages and discard duplicates.

A simple solution is for a message consumer to track the messages that it has processed using the `message id` and discard any duplicates. It could, for example, store the `message id` of each message that it consumed in a database table. Figure 3.12 shows how to do this using a dedicated table.

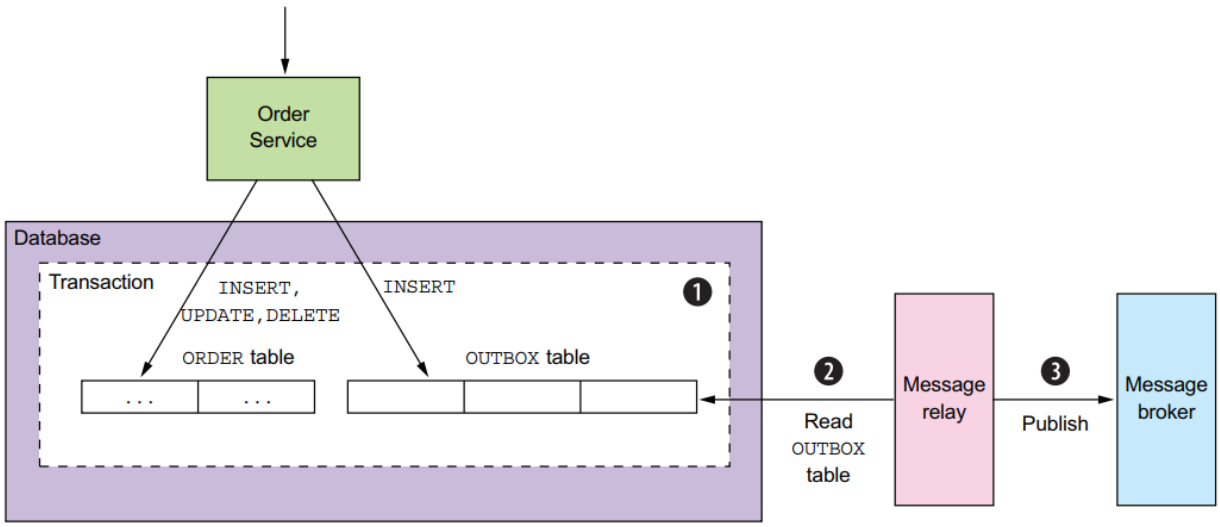


## Transactional messaging

A service often needs to publish messages as part of a transaction that updates the database. For instance, throughout this book you see examples of services that publish domain events whenever they create or update business entities. Both the database update and the sending of the message must happen within a transaction. Otherwise, a service might update the database and then crash, for example, before sending the message. If the service doesn't perform these two operations atomically, a failure could leave the system in an inconsistent state.

### USING A DATABASE TABLE AS A MESSAGE QUEUE

Let's imagine that your application is using a relational database. A straightforward way to reliably publish messages is to apply the Transactional outbox pattern. This pattern uses a database table as a temporary message queue. As figure 3.13 shows, a service that sends messages has an `OUTBOX` database table. As part of the database




## Pattern of getting message to publisher

#### PUBLISHING EVENTS BY USING THE POLLING PUBLISHER PATTERN

If the application uses a relational database, a very simple way to publish the messages inserted into the `OUTBOX` table is for the `MessageRelay` to poll the table for unpublished messages. It periodically queries the table:

```
SELECT * FROM OUTBOX ORDERED BY ... ASC
```

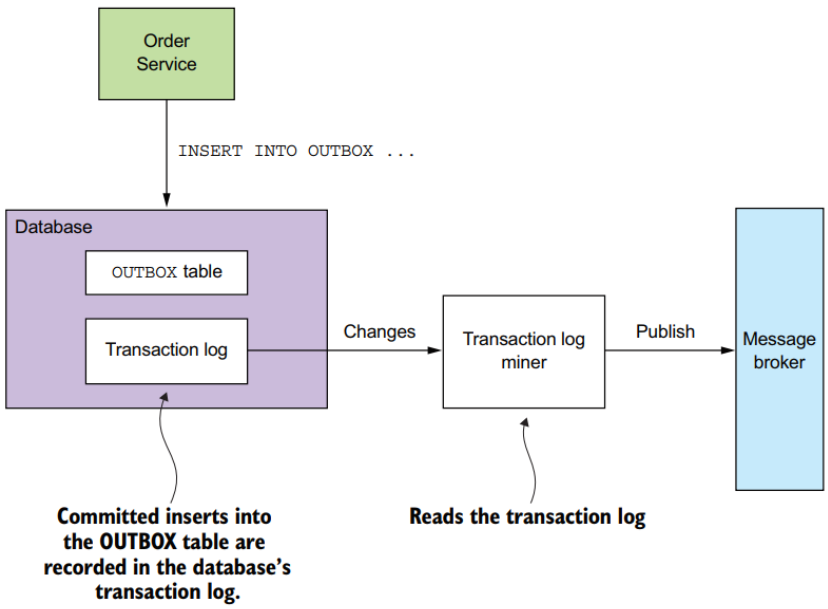
Next, the `MessageRelay` publishes those messages to the message broker, sending one to its destination message channel. Finally, it deletes those messages from the `OUTBOX` table:

```
BEGIN
 DELETE FROM OUTBOX WHERE ID in (....)
COMMIT
```

> **Pattern: Polling publisher**
> Publish messages by polling the outbox in the database. See http://microservices.io/patterns/data/polling-publisher.html.

#### PUBLISHING EVENTS BY APPLYING THE TRANSACTION LOG TAILING PATTERN

A sophisticated solution is for `MessageRelay` to *tail* the database transaction log (also called the commit log). Every committed update made by an application is represented as an entry in the database's transaction log. A transaction log miner can read the transaction log and publish each change as a message to the message broker. Figure 3.14 shows how this approach works.



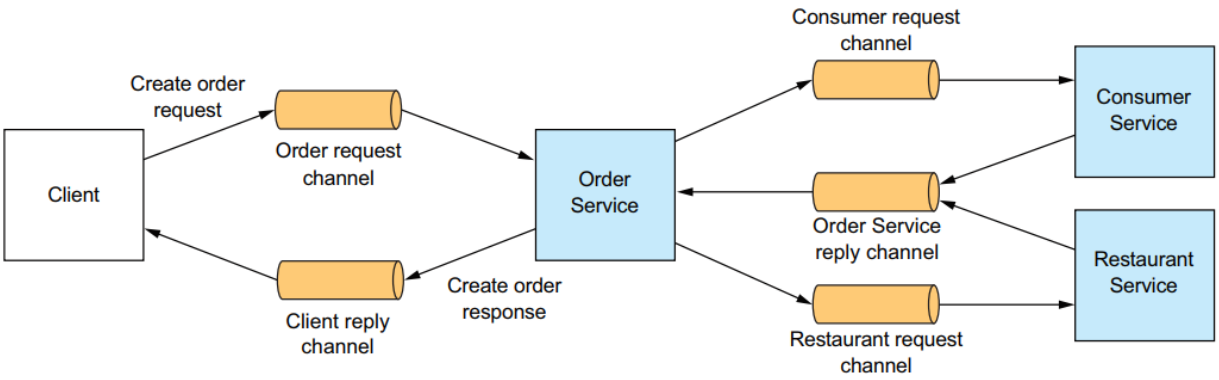

### 2.3.4 Using asynchronous messaging to improve availability

## Synchronous communication reduces availability

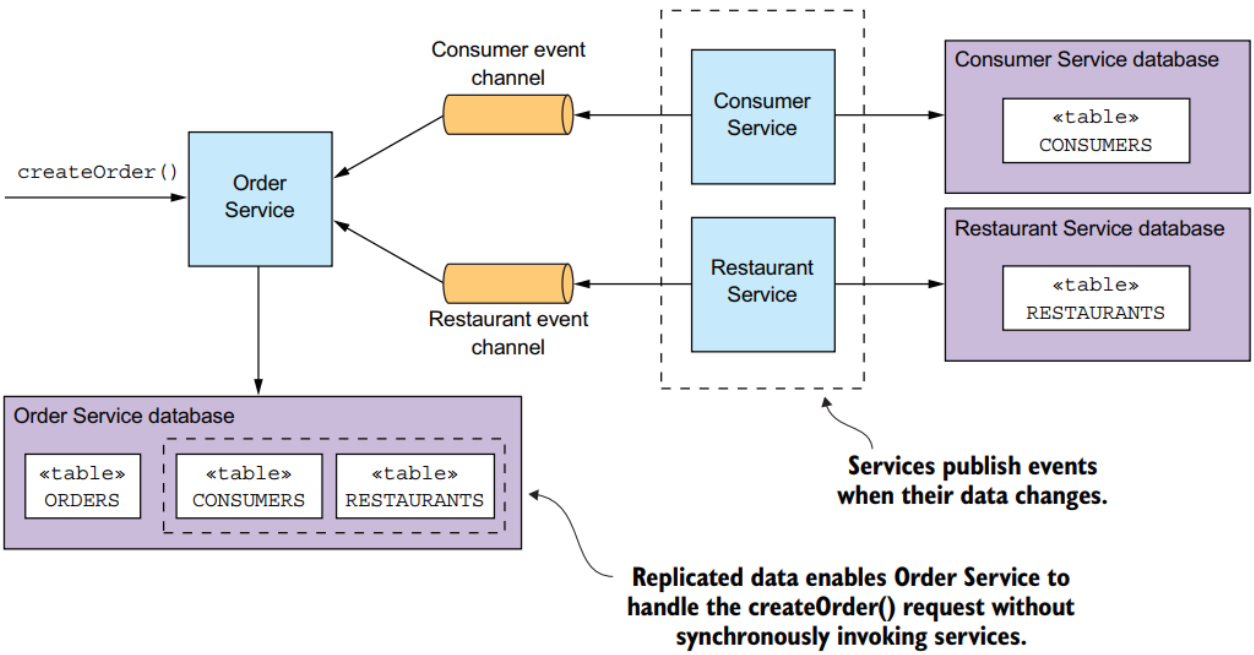
> Because these services use HTTP, they must all be simultaneously available in order for the FTGO application to process the `CreateOrder` request. The FTGO application couldn't create orders if any one of these three services is down. Mathematically speaking, the availability of a system operation is the product of the availability of the services that are invoked by that operation. If the `Order Service` and the two services that it invokes are 99.5% available, the overall availability is $99.5\%^3 = 98.5\%$, which is significantly less. Each additional service that participates in handling a request further reduces availability.

## Eliminating synchronous interaction

Using Async request




Replicate data

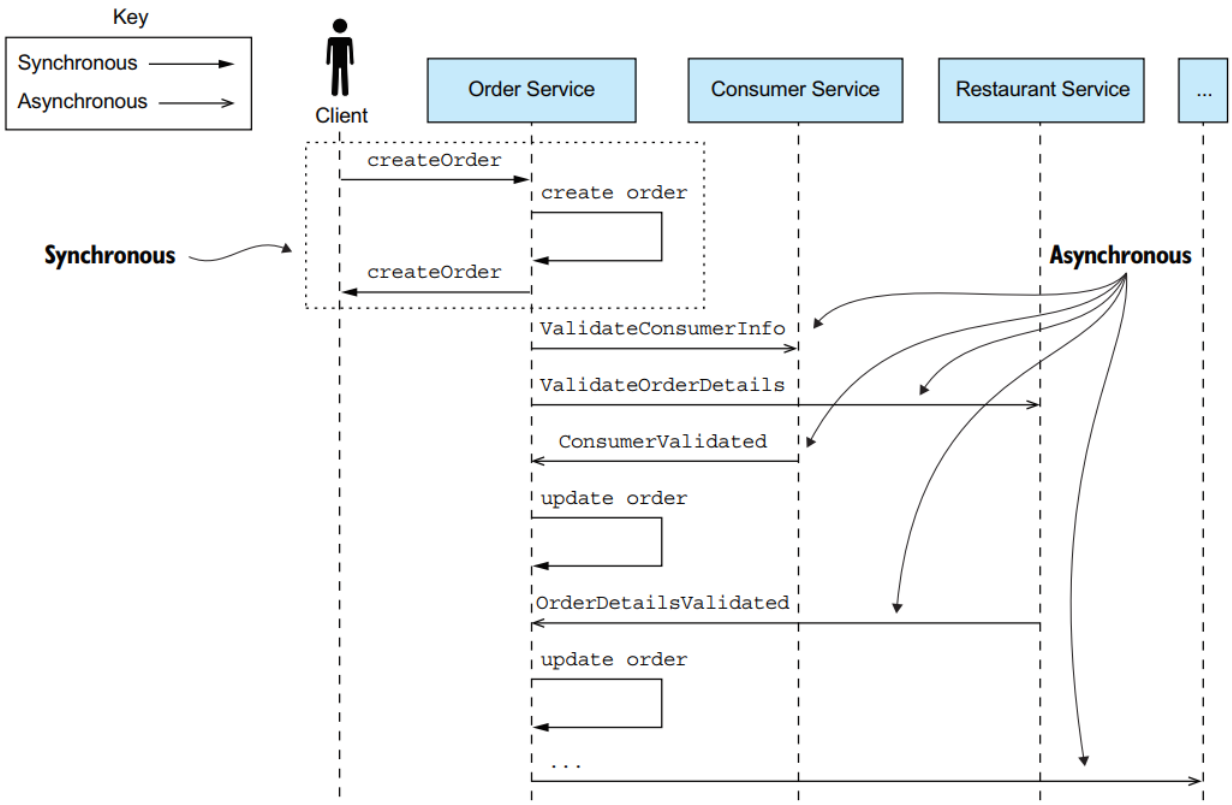



FINISH PROCESSING AFTER RETURNING A RESPONSE

#### FINISH PROCESSING AFTER RETURNING A RESPONSE

Another way to eliminate synchronous communication during request processing is for a service to handle a request as follows:

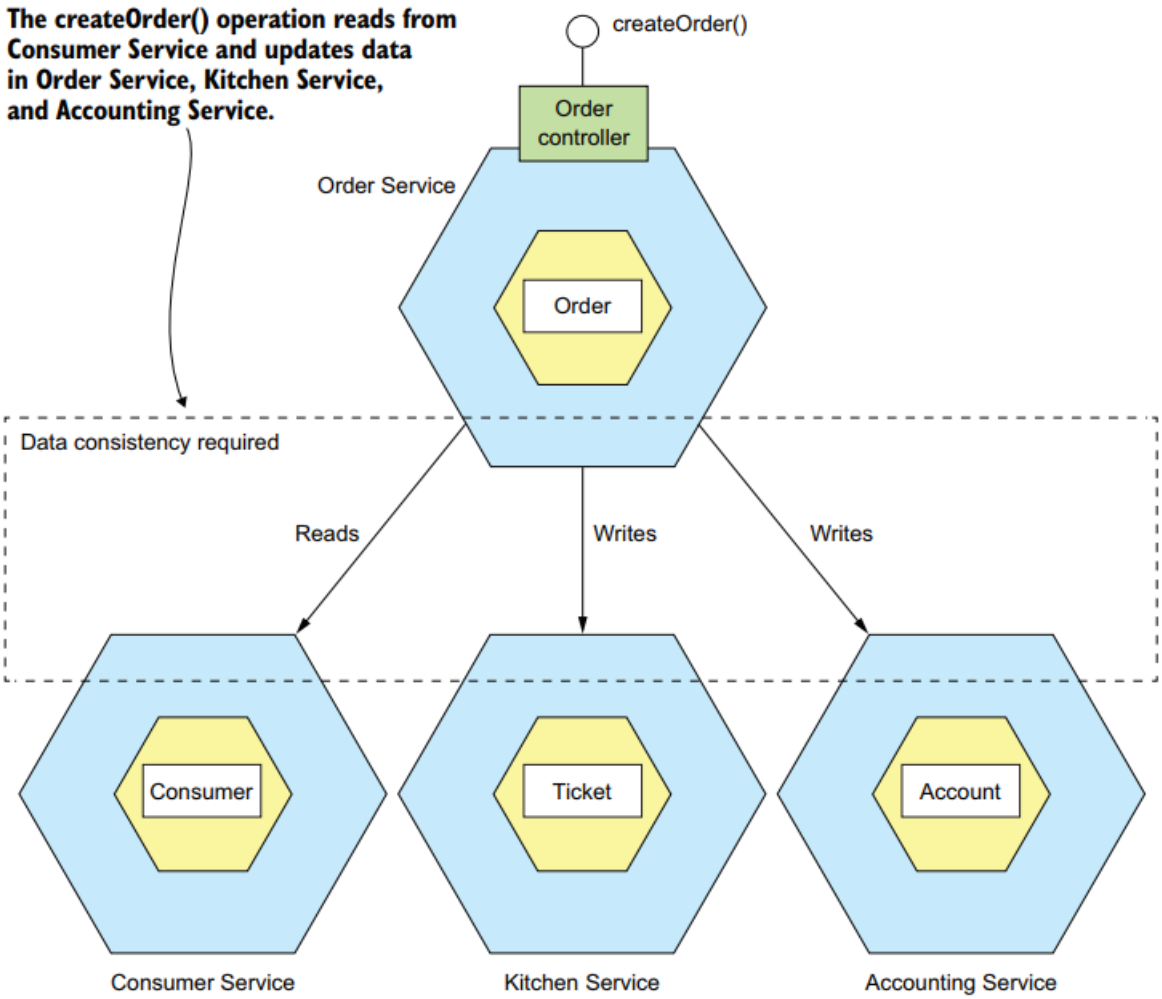
1. Validate the request using only the data available locally.
2. Update its database, including inserting messages into the `OUTBOX` table.
3. Return a response to its client.



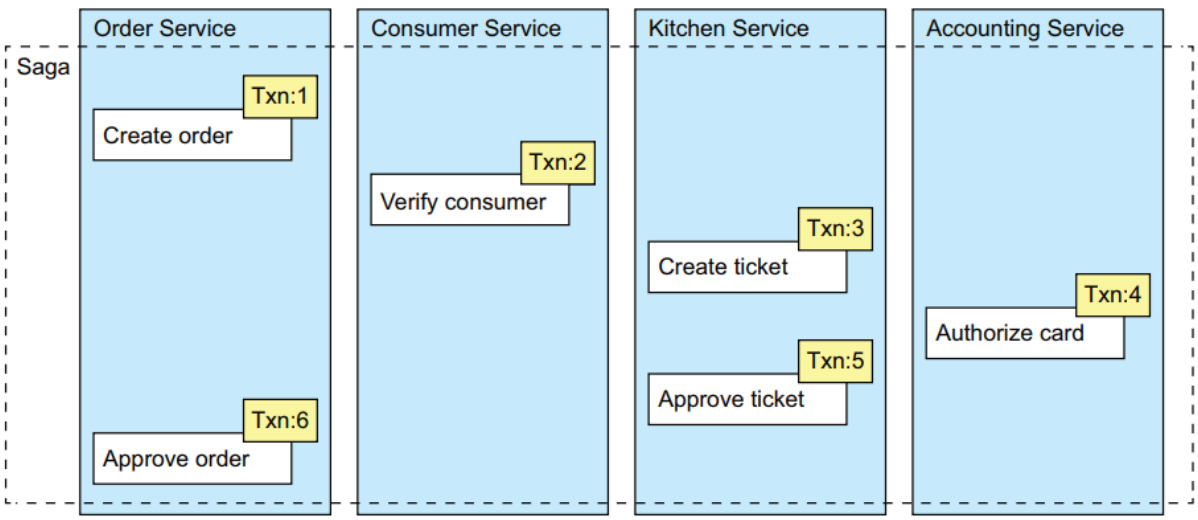

## 2.4 Sagas

### 2.4.1 Transaction in microservices

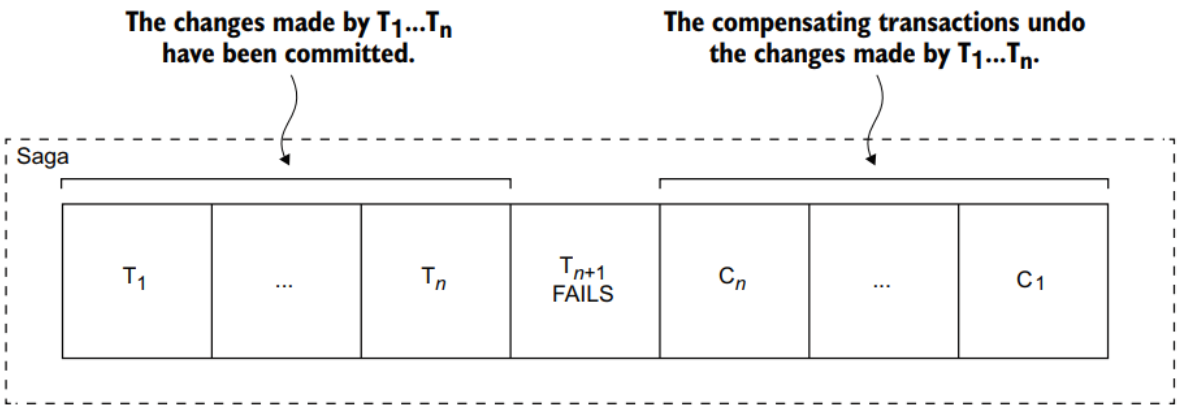
Transaction management in a microservice architecture




Sagas differ from ACID transactions in a couple of important ways. As I describe in detail in section 4.3, they lack the isolation property of ACID transactions. Also, because each local transaction commits its changes, a saga must be rolled back using compensating transactions. I talk more about compensating transactions later in this section. Let's take a look at an example saga.




Compensation commit:




If a local transaction fails, the saga's coordination mechanism must execute compensating transactions that reject the `Order` and possibly the `Ticket`. Table 4.1 shows the compensating transactions for each step of the `Create Order Saga`. It's important to note that not all steps need compensating transactions. Read-only steps, such as `verifyConsumerDetails()`, don't need compensating transactions. Nor do steps such as `authorizeCreditCard()` that are followed by steps that always succeed.

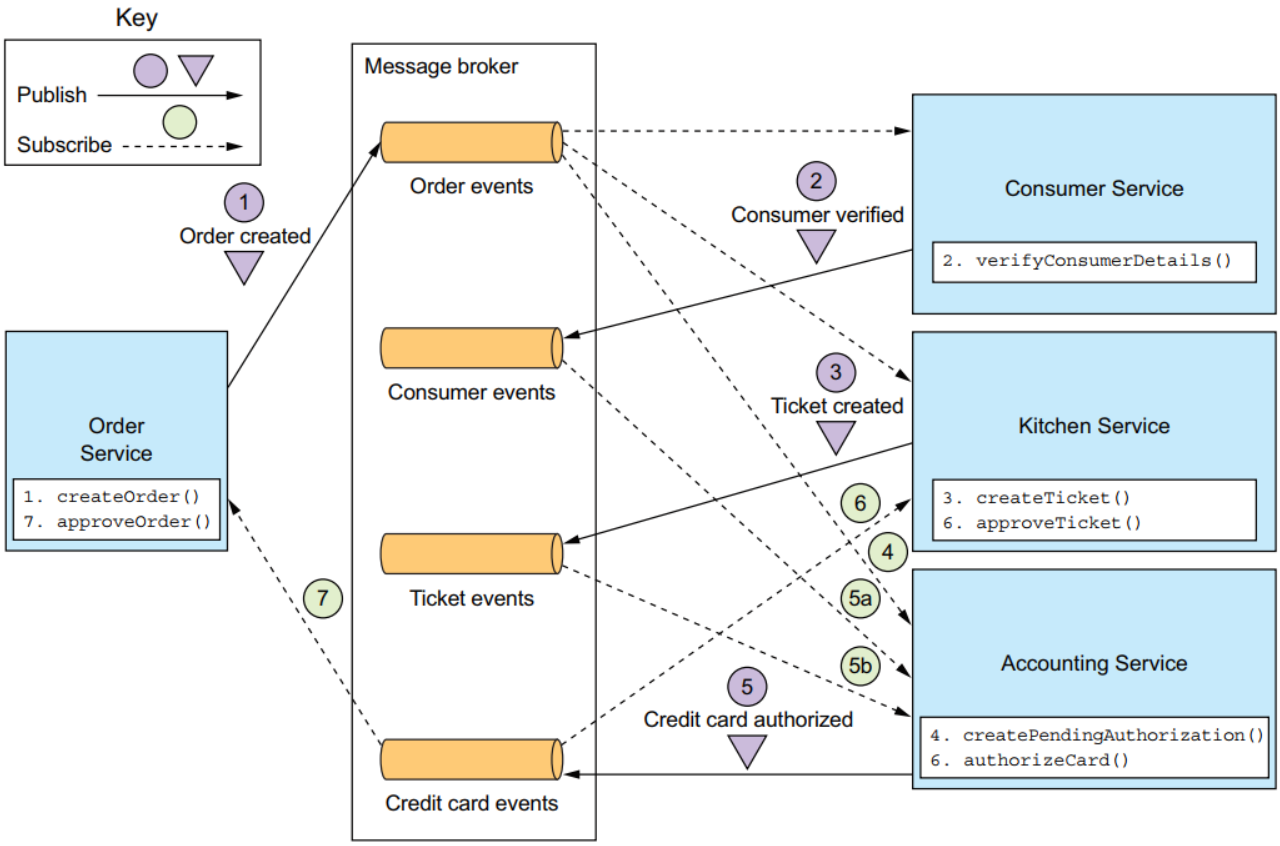
**Table 4.1** The compensating transactions for the `Create Order Saga`

| Step | Service | Transaction | Compensating transaction |
|---|---|---|---|
| 1 | `Order Service` | `createOrder()` | `rejectOrder()` |
| 2 | `Consumer Service` | `verifyConsumerDetails()` | — |
| 3 | `Kitchen Service` | `createTicket()` | `rejectTicket()` |
| 4 | `Accounting Service` | `authorizeCreditCard()` | — |
| 5 | `Kitchen Service` | `approveTicket()` | — |
| 6 | `Order Service` | `approveOrder()` | — |

### 2.4.2 Coordinating sagas

- *Choreography*—Distribute the decision making and sequencing among the saga participants. They primarily communicate by exchanging events.

- *Orchestration*—Centralize a saga's coordination logic in a saga orchestrator class. A saga *orchestrator* sends command messages to saga participants telling them which operations to perform.

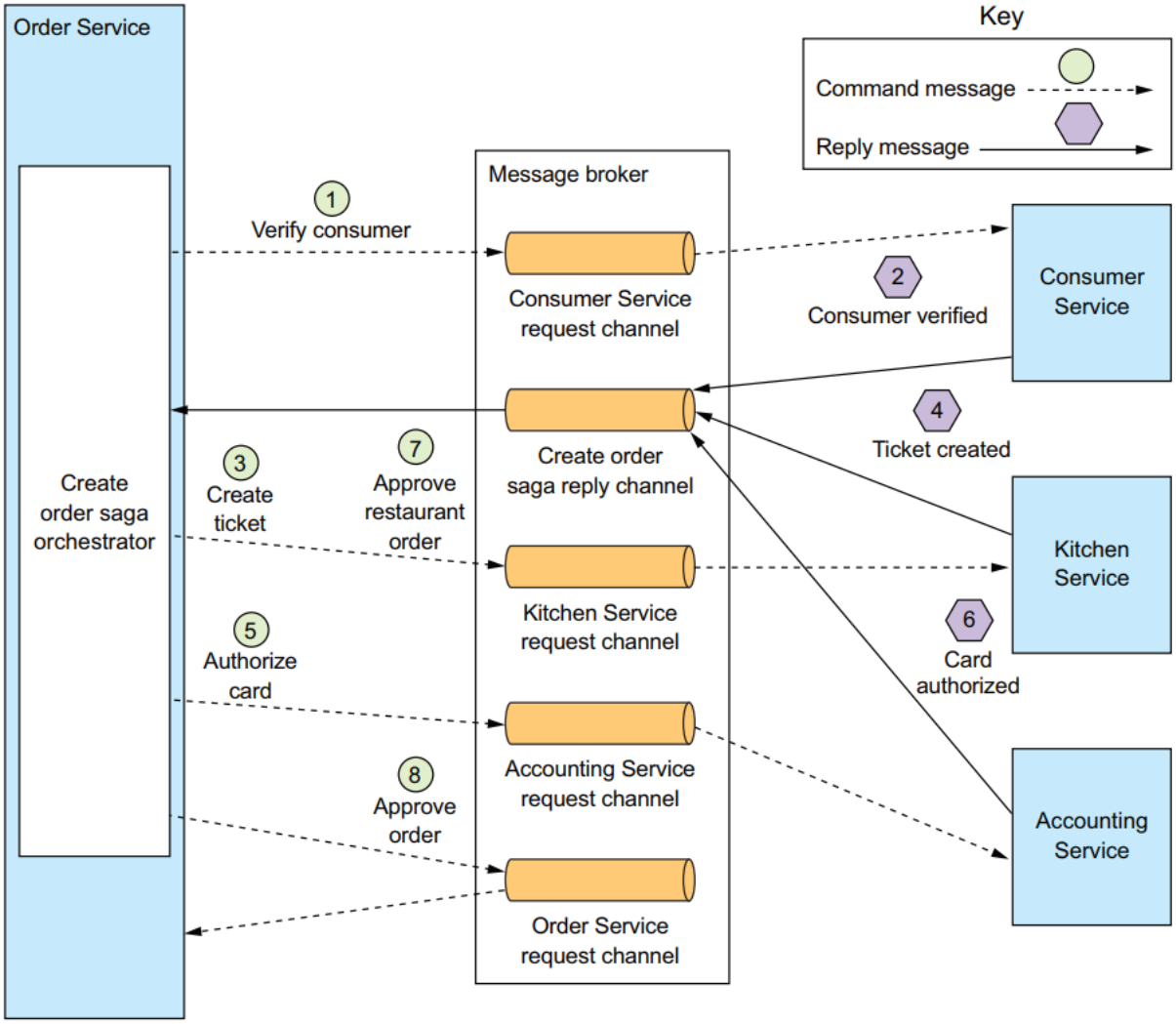



The happy path through this saga is as follows:

1. `Order Service` creates an `Order` in the `APPROVAL_PENDING` state and publishes an `OrderCreated` event.
2. `Consumer Service` consumes the `OrderCreated` event, verifies that the consumer can place the order, and publishes a `ConsumerVerified` event.
3. `Kitchen Service` consumes the `OrderCreated` event, validates the `Order`, creates a `Ticket` in a `CREATE_PENDING` state, and publishes the `TicketCreated` event.
4. `Accounting Service` consumes the `OrderCreated` event and creates a `CreditCardAuthorization` in a `PENDING` state.
5. `Accounting Service` consumes the `TicketCreated` and `ConsumerVerified` events, charges the consumer's credit card, and publishes the `CreditCardAuthorized` event.
6. `Kitchen Service` consumes the `CreditCardAuthorized` event and changes the state of the `Ticket` to `AWAITING_ACCEPTANCE`.
7. `Order Service` receives the `CreditCardAuthorized` events, changes the state of the `Order` to `APPROVED`, and publishes an `OrderApproved` event.

- First issue = ensure publish event = persist data
- Second issue = mapping received event to own data

## Orchestration-based sagas




1. The saga orchestrator sends a `Verify Consumer` command to `Consumer Service`.
2. `Consumer Service` replies with a `Consumer Verified` message.
3. The saga orchestrator sends a `Create Ticket` command to `Kitchen Service`.
4. `Kitchen Service` replies with a `Ticket Created` message.
5. The saga orchestrator sends an `Authorize Card` message to `Accounting Service`.
6. `Accounting Service` replies with a `Card Authorized` message.
7. The saga orchestrator sends an `Approve Ticket` command to `Kitchen Service`.
8. The saga orchestrator sends an `Approve Order` command to `Order Service`.

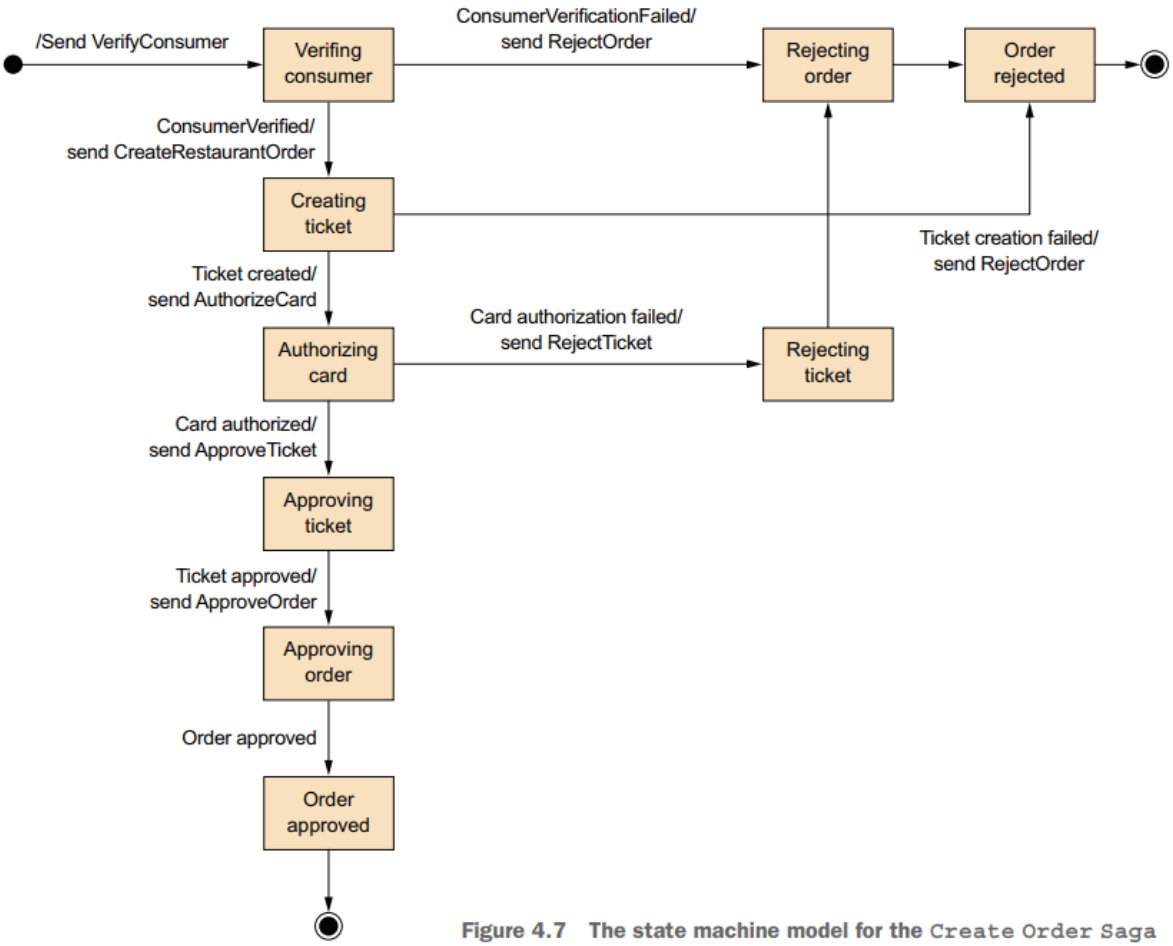



**Figure 4.7 The state machine model for the `Create Order Saga`**

### 2.4.3 Handling the lack of isolation

On the surface, the lack of isolation sounds unworkable. But in practice, it's common for developers to accept reduced isolation in return for higher performance. An RDBMS lets you specify the isolation level for each transaction (https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html). The default isolation level is usually an isolation level that's weaker than full isolation, also known as serializable transactions. Real-world database transactions are often different from textbook definitions of ACID transactions.

#### *Overview of anomalies*

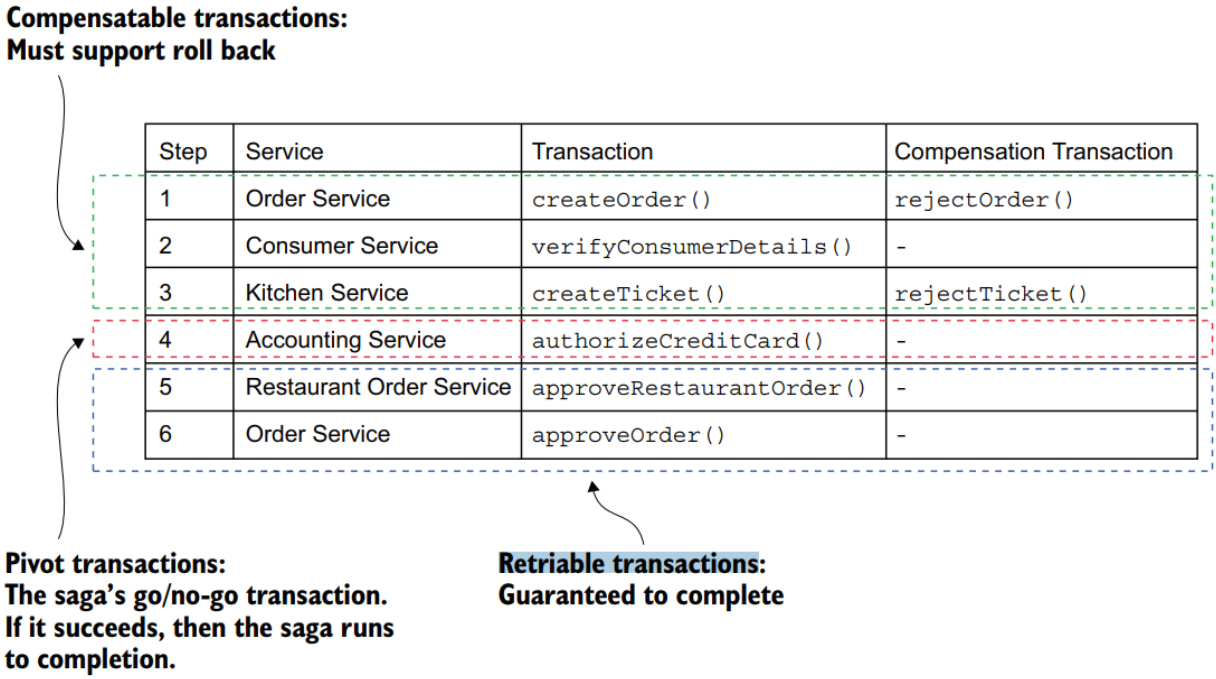
The lack of isolation can cause the following three anomalies:

- *Lost updates*—One saga overwrites without reading changes made by another saga.
- *Dirty reads*—A transaction or a saga reads the updates made by a saga that has not yet completed those updates.
- *Fuzzy/nonrepeatable reads*—Two different steps of a saga read the same data and get different results because another saga has made updates.

Countermeasures for handling the lack of isolation

Type of transaction in sagas:

- Compensatable transaction
- Pivot transaction
- Retriable transactions

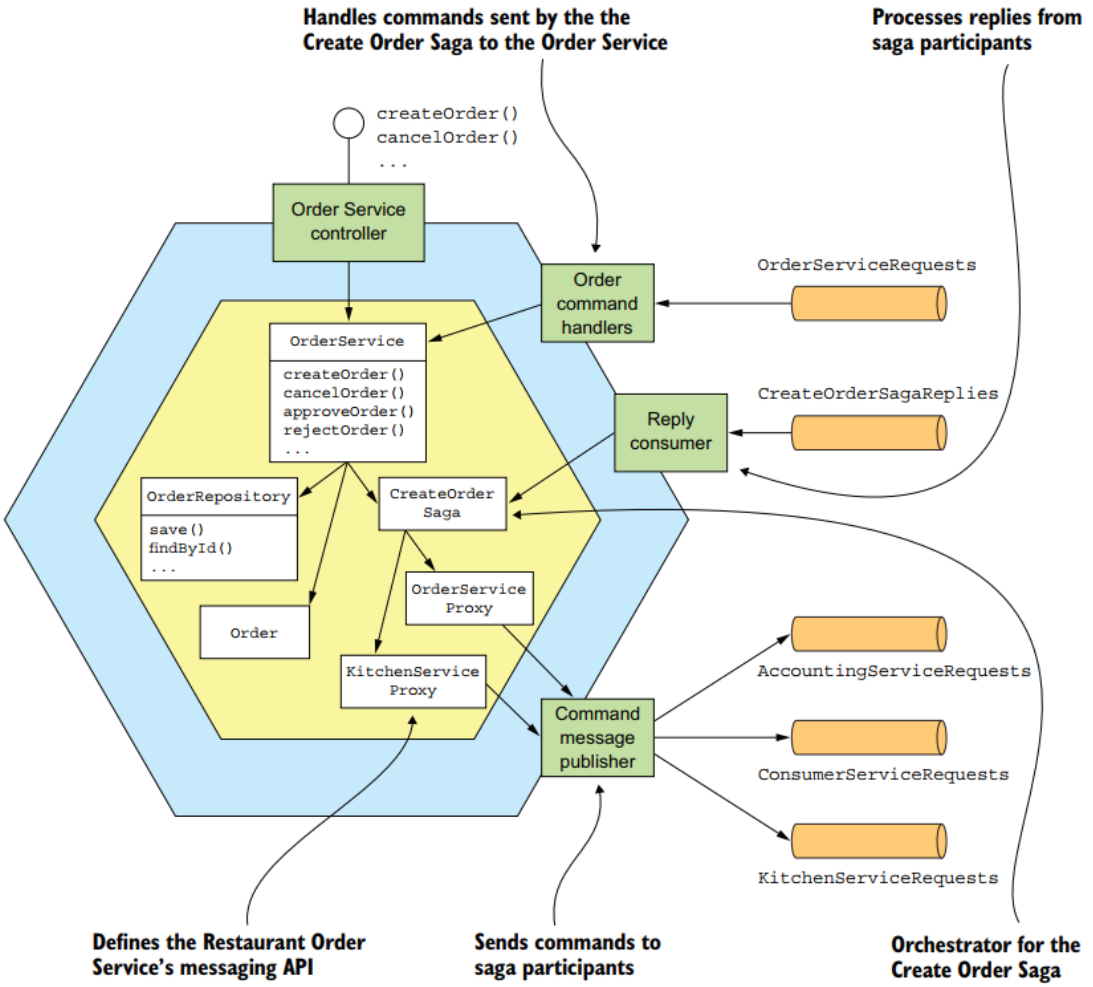



| Step | Service | Transaction | Compensation Transaction |
|---|---|---|---|
| 1 | Order Service | `createOrder()` | `rejectOrder()` |
| 2 | Consumer Service | `verifyConsumerDetails()` | - |
| 3 | Kitchen Service | `createTicket()` | `rejectTicket()` |
| 4 | Accounting Service | `authorizeCreditCard()` | - |
| 5 | Restaurant Order Service | `approveRestaurantOrder()` | - |
| 6 | Order Service | `approveOrder()` | - |

Counter measuring:

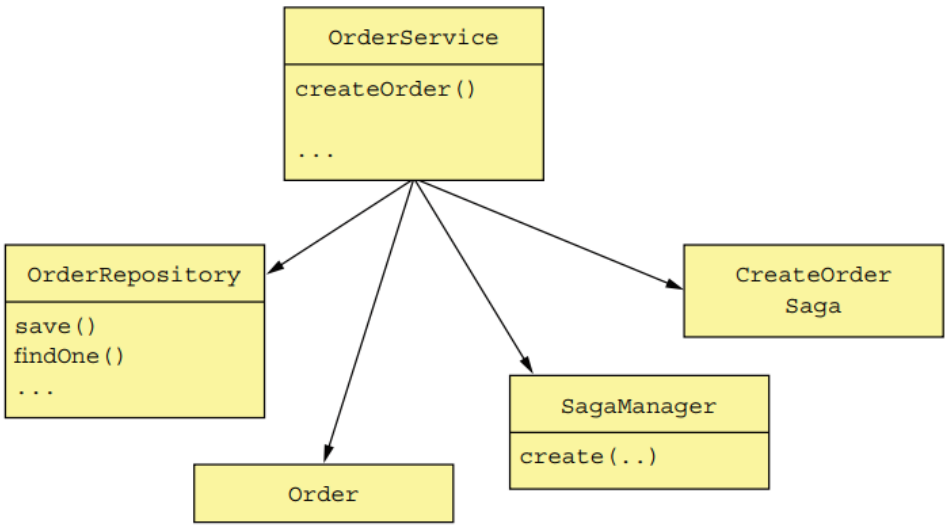
- *Semantic lock*—An application-level lock.
- *Commutative updates*—Design update operations to be executable in any order.
- *Pessimistic view*—Reorder the steps of a saga to minimize business risk.
- *Reread value*—Prevent dirty writes by rereading data to verify that it's unchanged before overwriting it.
- *Version file*—Record the updates to a record so that they can be reordered.
- *By value*—Use each request's business risk to dynamically select the concurrency mechanism.

Order service:

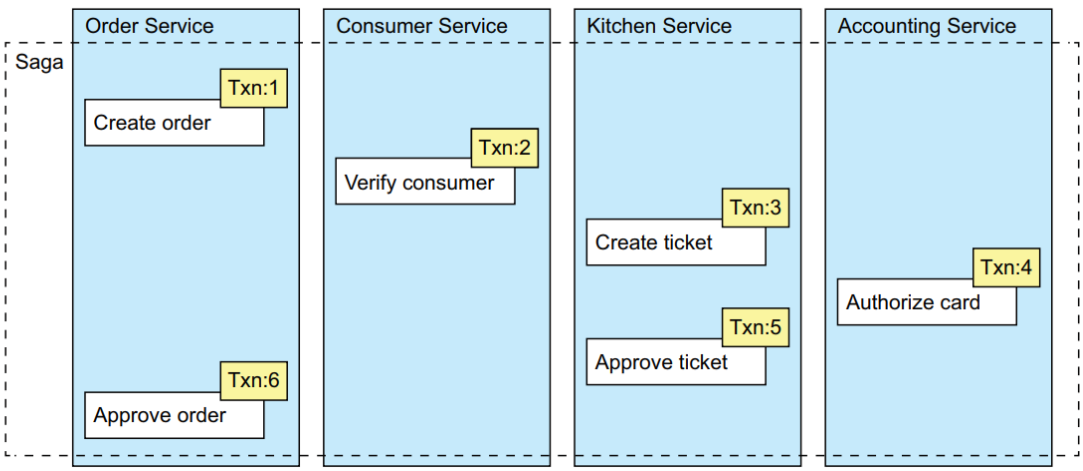

Handles commands sent by the the Create Order Saga to the Order Service
Processes replies from saga participants
createOrder()
cancelOrder()
...
Order Service controller
Order command handlers
OrderServiceRequests
OrderService
createOrder()
cancelOrder()
approveOrder()
rejectOrder()
...
Reply consumer
CreateOrderSagaReplies
OrderRepository
save()
findById()
...
CreateOrder Saga
OrderService Proxy
Order
KitchenService Proxy
Command message publisher
AccountingServiceRequests
ConsumerServiceRequests
KitchenServiceRequests
Defines the Restaurant Order Service's messaging API
Sends commands to saga participants
Orchestrator for the Create Order Saga

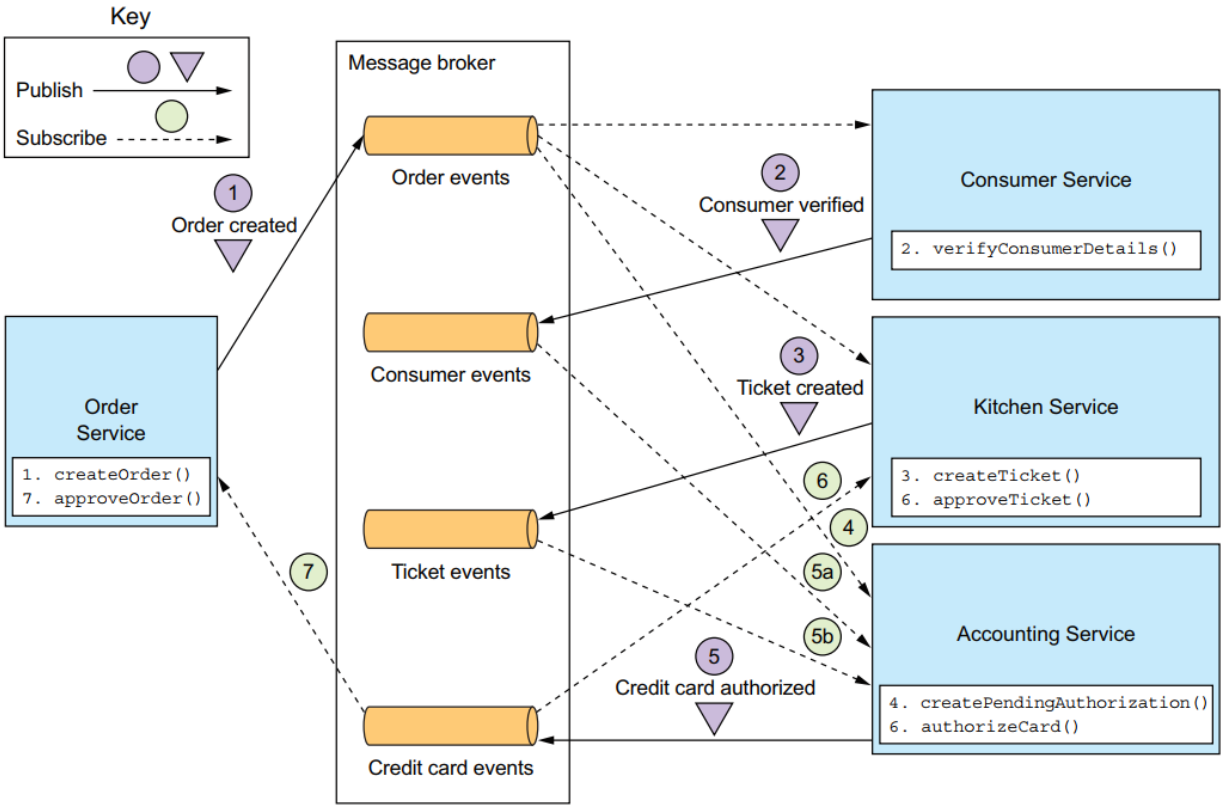
### *The OrderService class*

The `OrderService` class is a domain service called by the service's API layer. It's responsible for creating and managing orders. Figure 4.10 shows `OrderService` and some of its collaborators. `OrderService` creates and updates `Orders`, invokes the `OrderRepository` to persist `Orders`, and creates sagas, such as the `CreateOrderSaga`, using the `SagaManager`. The `SagaManager` class is one of the classes provided by the Eventuate Tram Saga framework, which is a framework for writing saga orchestrators and participants, and is discussed a little later in this section.

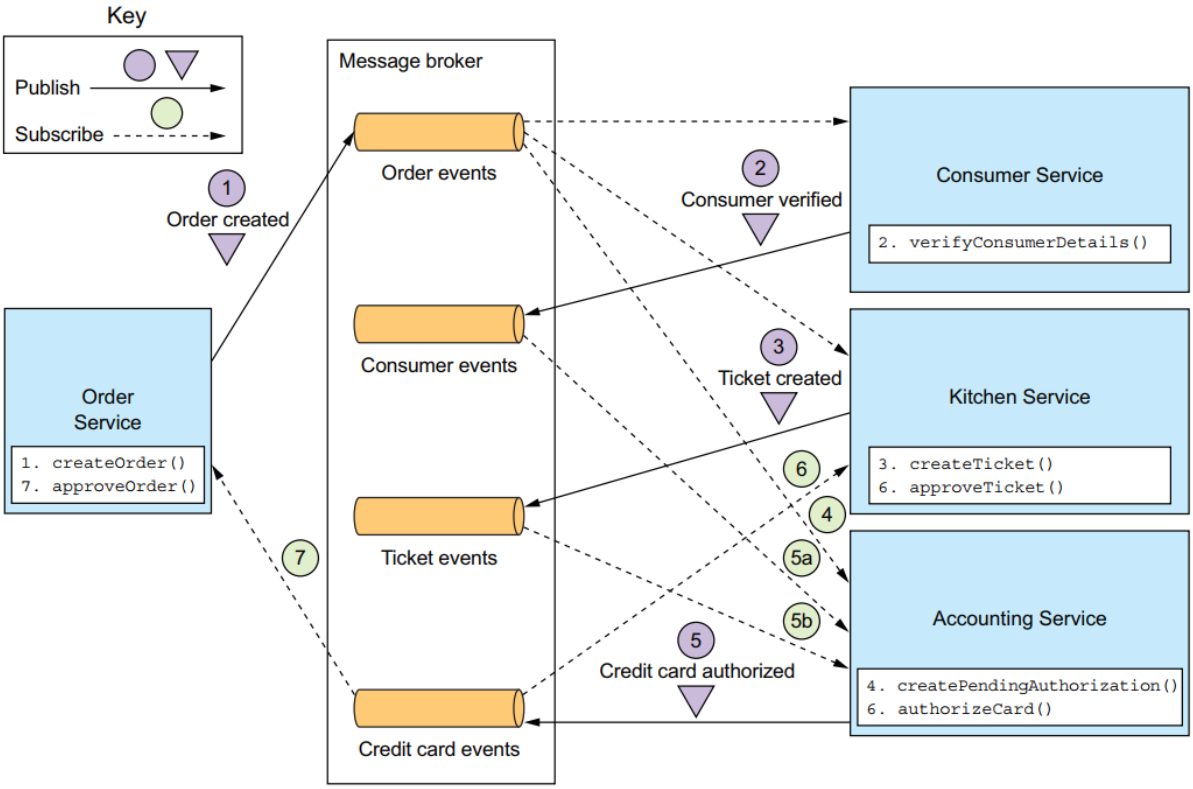



## Implemenetation of Saga:





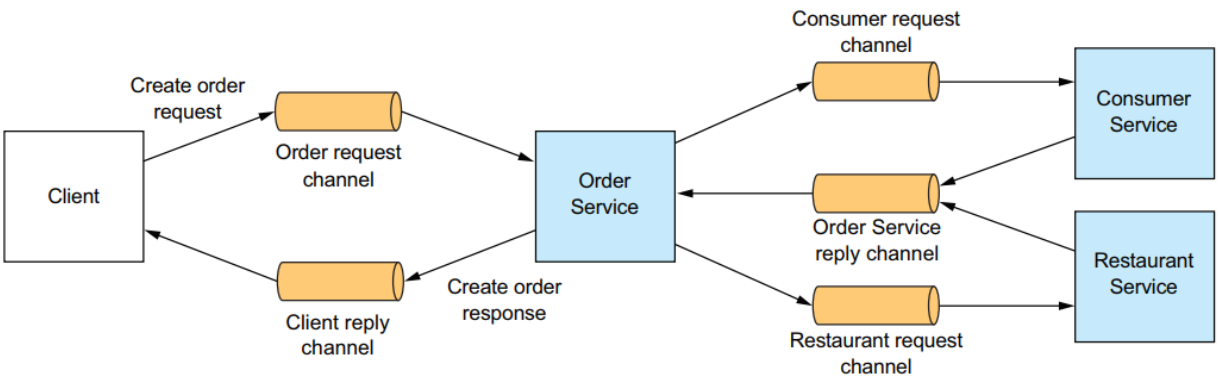

1. `Order Service` creates an `Order` in the `APPROVAL_PENDING` state and publishes an `OrderCreated` event.
2. `Consumer Service` consumes the `OrderCreated` event, verifies that the consumer can place the order, and publishes a `ConsumerVerified` event.
3. `Kitchen Service` consumes the `OrderCreated` event, validates the `Order`, creates a `Ticket` in a `CREATE_PENDING` state, and publishes the `TicketCreated` event.
4. `Accounting Service` consumes the `OrderCreated` event and creates a `CreditCardAuthorization` in a `PENDING` state.
5. `Accounting Service` consumes the `TicketCreated` and `ConsumerVerified` events, charges the consumer's credit card, and publishes a `Credit Card Authorization Failed` event.
6. `Kitchen Service` consumes the `Credit Card Authorization Failed` event and changes the state of the `Ticket` to `REJECTED`.
7. `Order Service` consumes the `Credit Card Authorization Failed` event and changes the state of the `Order` to `REJECTED`.


Key
Publish
Subscribe
Message broker
Order events
Consumer events
Ticket events
Credit card events
1
Order created
Order
Service
1. createOrder()
7. approveOrder()
2
Consumer verified
Consumer Service
2. verifyConsumerDetails()
3
Ticket created
Kitchen Service
3. createTicket()
6. approveTicket()
4
5a
5b
6
7
5
Credit card authorized
Accounting Service
4. createPendingAuthorization()
6. authorizeCard()

1. The saga orchestrator sends a `Verify Consumer` command to `Consumer Service`.
2. `Consumer Service` replies with a `Consumer Verified` message.
3. The saga orchestrator sends a `Create Ticket` command to `Kitchen Service`.
4. `Kitchen Service` replies with a `Ticket Created` message.
5. The saga orchestrator sends an `Authorize Card` message to `Accounting Service`.
6. `Accounting Service` replies with a `Card Authorized` message.
7. The saga orchestrator sends an `Approve Ticket` command to `Kitchen Service`.
8. The saga orchestrator sends an `Approve Order` command to `Order Service`.

The sequence of events is as follows:

1. `Order Service` creates an `Order` in the `APPROVAL_PENDING` state and publishes an `OrderCreated` event.
2. `Consumer Service` consumes the `OrderCreated` event, verifies that the consumer can place the order, and publishes a `ConsumerVerified` event.
3. `Kitchen Service` consumes the `OrderCreated` event, validates the `Order`, creates a `Ticket` in a `CREATE_PENDING` state, and publishes the `TicketCreated` event.
4. `Accounting Service` consumes the `OrderCreated` event and creates a `CreditCardAuthorization` in a `PENDING` state.
5. `Accounting Service` consumes the `TicketCreated` and `ConsumerVerified` events, charges the consumer's credit card, and publishes a `Credit Card Authorization Failed` event.
6. `Kitchen Service` consumes the `Credit Card Authorization Failed` event and changes the state of the `Ticket` to `REJECTED`.
7. `Order Service` consumes the `Credit Card Authorization Failed` event and changes the state of the `Order` to `REJECTED`.




## 2.5 Business Logic in Microservices

Fortunately, we can address these issues by using the Aggregate pattern from DDD. The Aggregate pattern structures a service's business logic as a collection of aggregates. An *aggregate* is a cluster of objects that can be treated as a unit. There are two reasons why aggregates are useful when developing business logic in a microservice architecture:

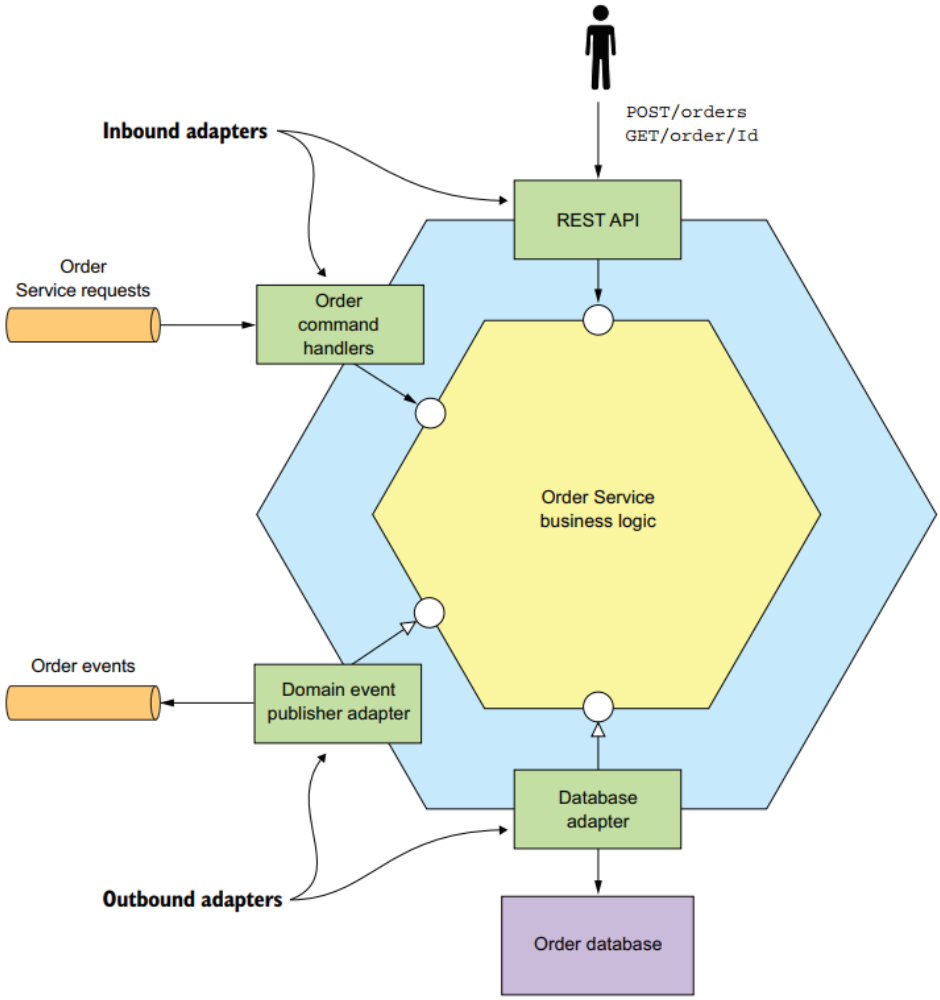
- Aggregates avoid any possibility of object references spanning service boundaries, because an inter-aggregate reference is a primary key value rather than an object reference.
- Because a transaction can only create or update a single aggregate, aggregates fit the constraints of the microservices transaction model.

### 2.5.1 Business logic organization patterns

Figure 5.1 shows the architecture of a typical service. As described in chapter 2, the business logic is the core of a hexagonal architecture. Surrounding the business logic are the inbound and outbound adapters. An *inbound adapter* handles requests from clients and invokes the business logic. An *outbound adapter*, which is invoked by the business logic, invokes other services and applications.

This service consists of the business logic and the following adapters:

- `REST API adapter`—An inbound adapter that implements a REST API which invokes the business logic
- `OrderCommandHandlers`—An inbound adapter that consumes command messages from a message channel and invokes the business logic
- `Database Adapter`—An outbound adapter that's invoked by the business logic to access the database
- `Domain Event Publishing Adapter`—An outbound adapter that publishes events to a message broker




There are two way to implement business logic:

- **Procedural Transaction script pattern**
- **Object-oriented domain model pattern**

Designing business logic using the Transaction script pattern

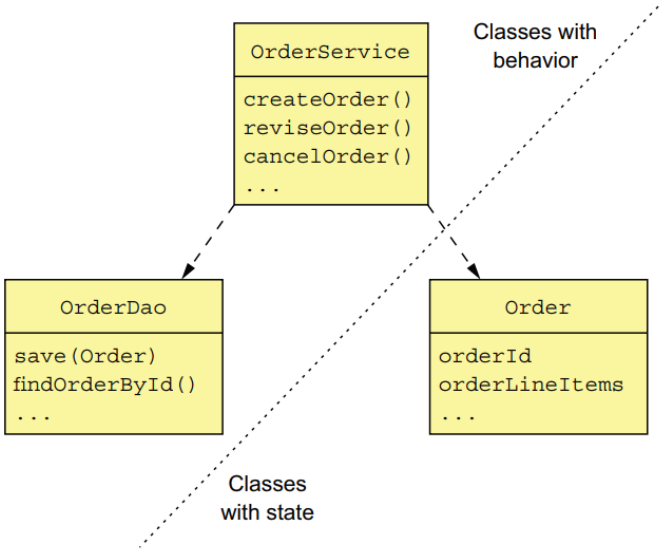
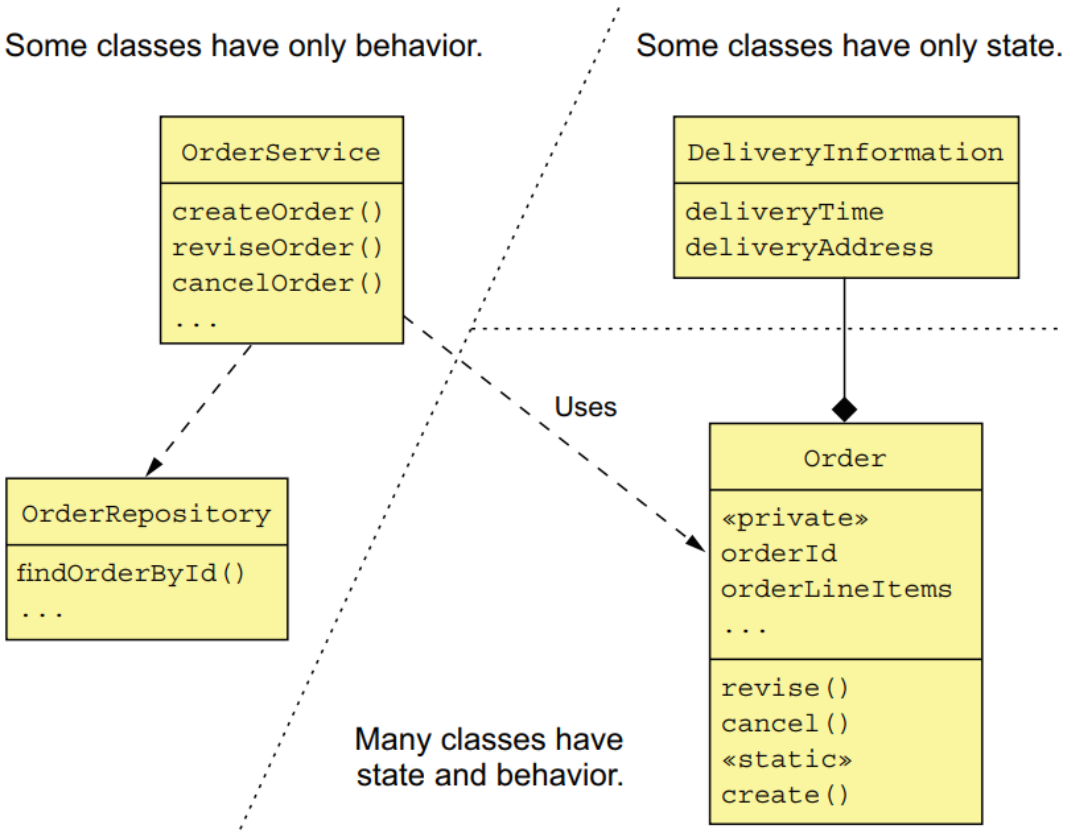



**Figure 5.2 Organizing business logic as transaction scripts. In a typical transaction script–based design, one set of classes implements behavior and another set stores state. The transaction scripts are organized into classes that typically have no state. The scripts use data classes, which typically have no behavior.**

### 2.5.2 Designing business logic using the Domain model pattern




### 2.5.3 Domain driven design

- *Entity*—An object that has a persistent identity. Two entities whose attributes have the same values are still different objects. In a Java EE application, classes that are persisted using JPA `@Entity` are usually DDD entities.
- *Value object*—An object that is a collection of values. Two value objects whose attributes have the same values can be used interchangeably. An example of a value object is a `Money` class, which consists of a currency and an amount.
- *Factory*—An object or method that implements object creation logic that's too complex to be done directly by a constructor. It can also hide the concrete classes that are instantiated. A factory might be implemented as a static method of a class.
- *Repository*—An object that provides access to persistent entities and encapsulates the mechanism for accessing the database.
- *Service*—An object that implements business logic that doesn't belong in an entity or a value object.

### 2.5.4 Aggregates have explicit boundaries

An *aggregate* is a cluster of domain objects within a boundary that can be treated as a unit. It consists of a root entity and possibly one or more other entities and value objects. Many business objects are modeled as aggregates. For example, in chapter 2

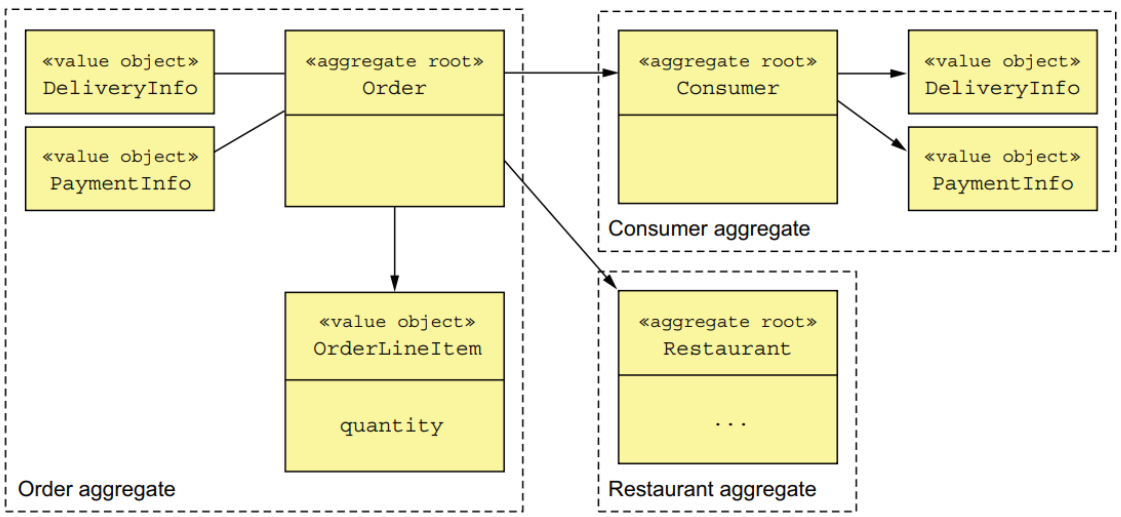
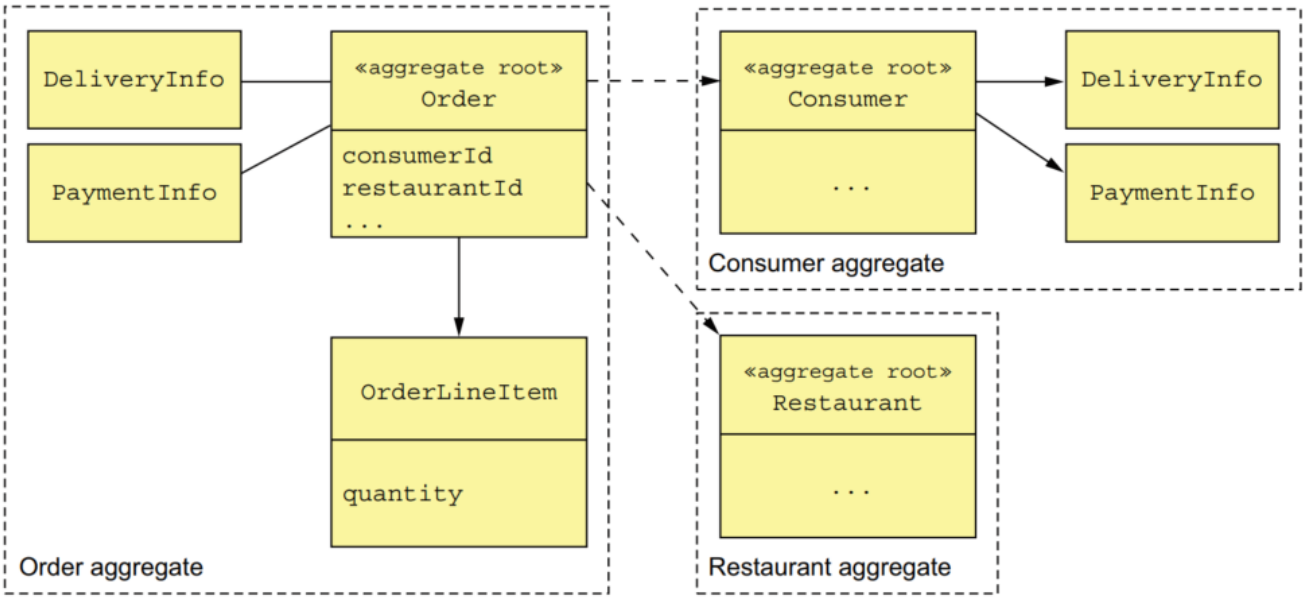



**Figure 5.5 Structuring a domain model as a set of aggregates makes the boundaries explicit.**

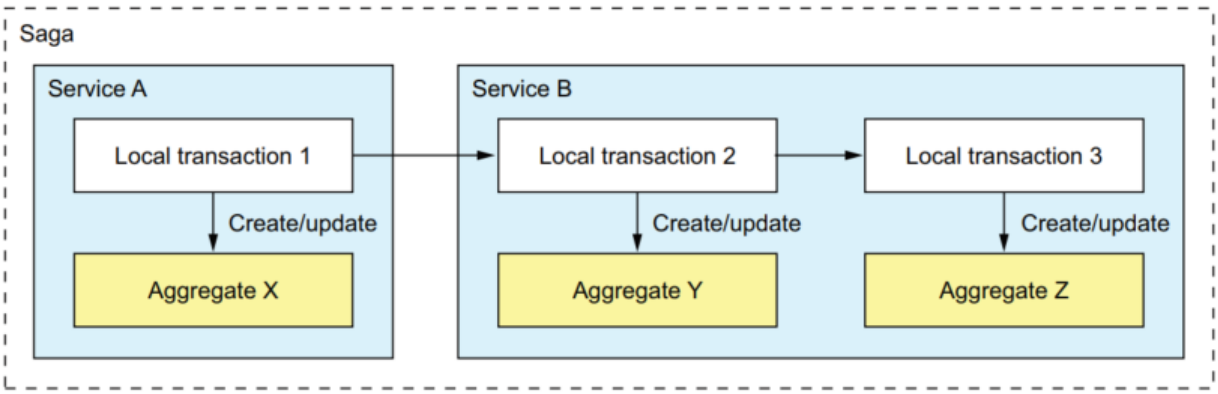
Aggrerate rule:

- REFERENCE ONLY THE AGGREGATE ROOT

- INTER-AGGREGATE REFERENCES MUST USE PRIMARY KEYS




- ONE TRANSACTION CREATES OR UPDATES ONE AGGREGATE



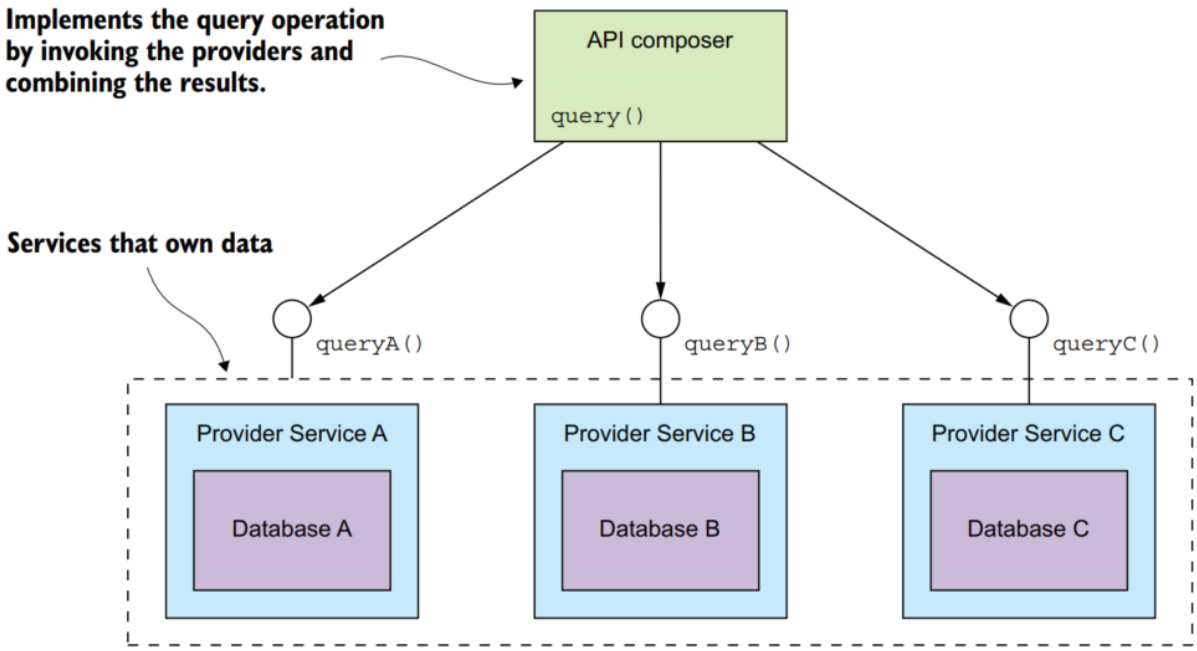

## 2.6 Implementing queries in a microservice architecture

There are two different patterns for implementing query operations in a microservice architecture:

- *The API composition pattern*—This is the simplest approach and should be used whenever possible. It works by making clients of the services that own the data responsible for invoking the services and combining the results.
- *The Command query responsibility segregation (CQRS) pattern*—This is more powerful than the API composition pattern, but it's also more complex. It maintains one or more view databases whose sole purpose is to support queries.

### 2.6.1 Querying using the API composition pattern

- *An API composer*—This implements the query operation by querying the provider services.
- *A provider service*—This is a service that owns some of the data that the query returns.




## API composition design issues

API Gateway responsible for aggregate gateway

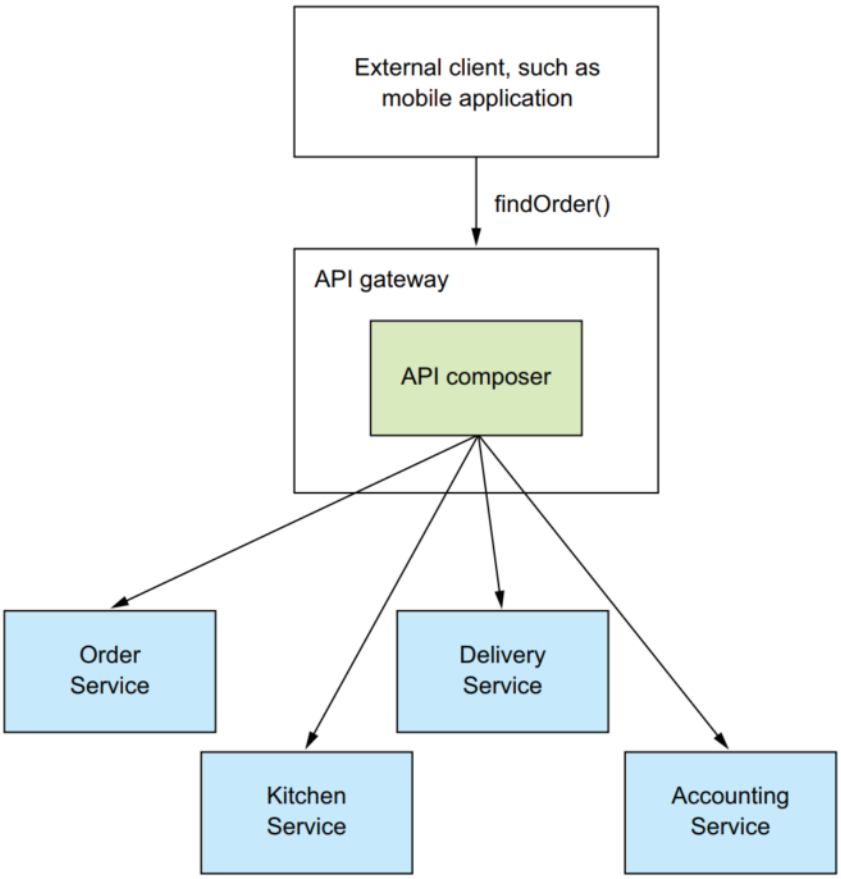



**Figure 7.5 Implementing API composition in the API gateway. The API queries the provider services to retrieve the data, combines the results, and returns a response to the client.**

- Query should be call in reactive model.

### 2.6.2 Using the CQRS pattern

Many enterprise applications use an RDBMS as the transactional system of record and a text search database, such as Elasticsearch or Solr, for text search queries. Some applications keep the databases synchronized by writing to both simultaneously. Others periodically copy data from the RDBMS to the text search engine. Applications with this architecture leverage the strengths of multiple databases: the transactional properties of the RDBMS and the querying capabilities of the text database.

#### Motivations for using CQRS

#### IMPLEMENTING THE FINDORDERHISTORY() QUERY OPERATION

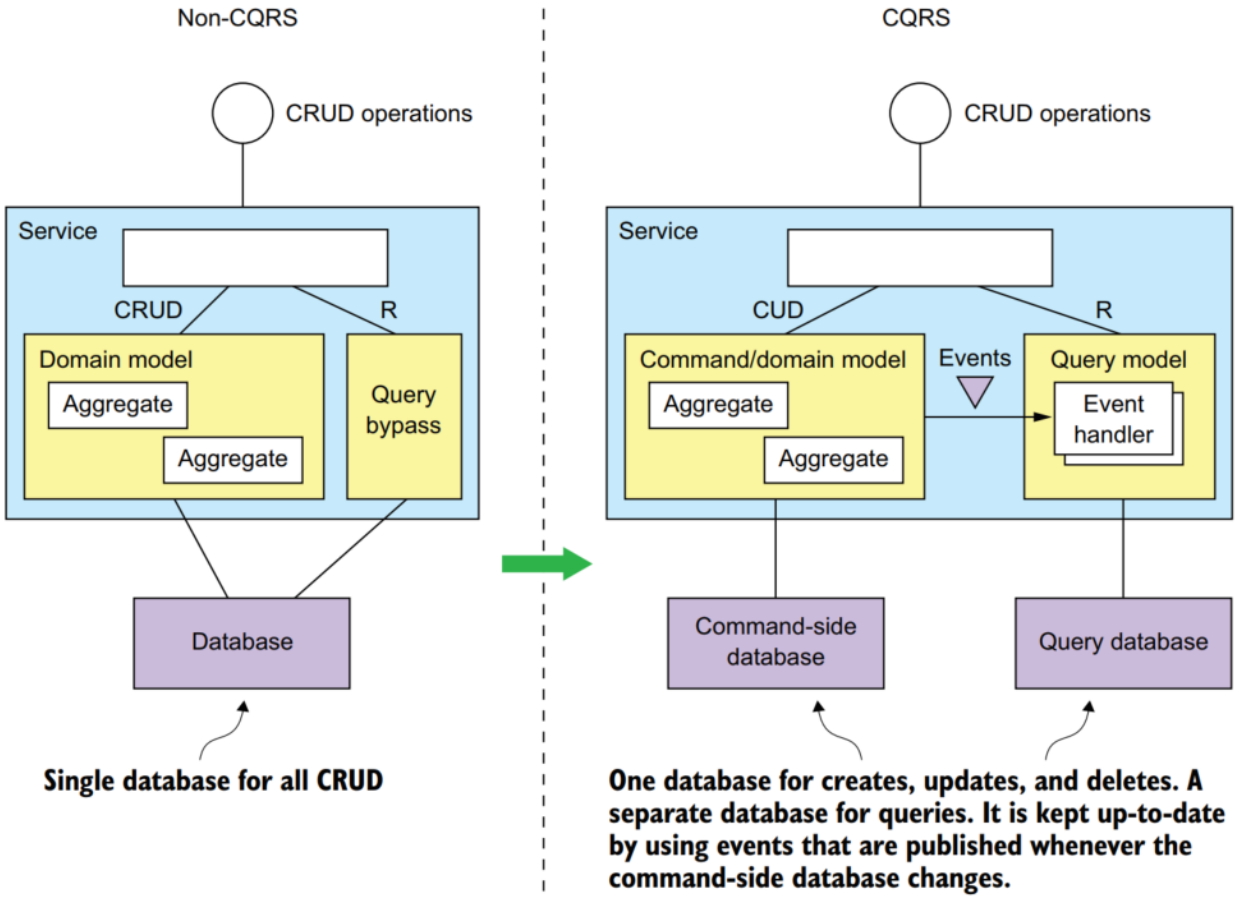
The `findOrderHistory()` operation retrieves a consumer's order history. It has several parameters:

- `consumerId`—Identifies the consumer
- `pagination`—Page of results to return
- `filter`—Filter criteria, including the max age of the orders to return, an optional order status, and optional keywords that match the restaurant name and menu items

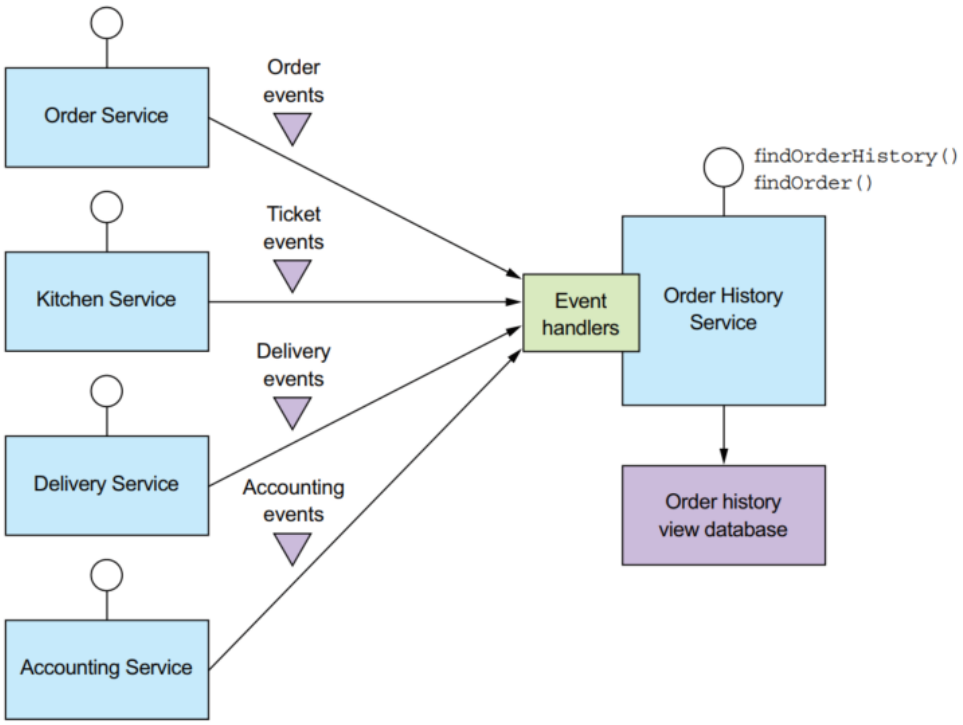
The challenge with using replicas is keeping them up-to-date whenever the original data changes. As you'll learn below, CQRS solves the problem of synchronizing replicas.

## Overview of CQRS

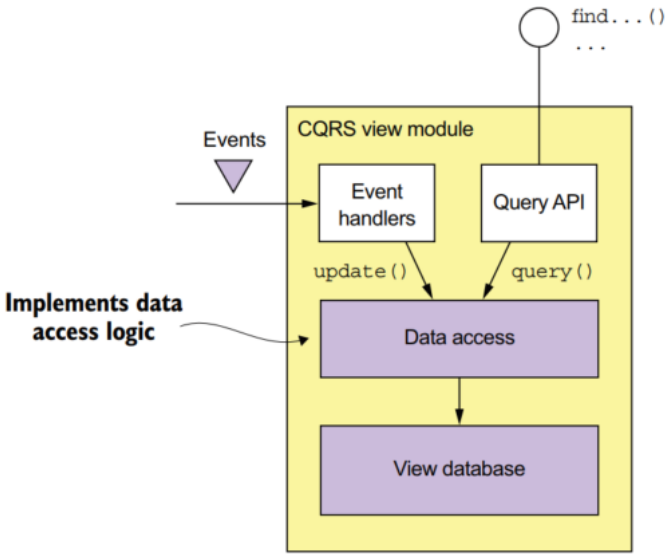
- CQRS SEPARATES COMMANDS FROM QUERIES




## Standalone query service:




### 2.6.3 Designing CQRS views




## Choosing a view datastore

| If you need | Use | Example |
|---|---|---|
| PK-based lookup of JSON objects | A document store such as MongoDB or DynamoDB, or a key value store such as Redis | Implement order history by maintaining a MongoDB document containing the per-customer. |
| Query-based lookup of JSON objects | A document store such as MongoDB or DynamoDB | Implement customer view using MongoDB or DynamoDB. |
| Text queries | A text search engine such as Elasticsearch | Implement text search for orders by maintaining a per-order Elasticsearch document. |
| Graph queries | A graph database such as Neo4j | Implement fraud detection by maintaining a graph of customers, orders, and other data. |
| Traditional SQL reporting/BI | An RDBMS | Standard business reports and analytics. |

SUPPORTING UPDATE OPERATIONS

- Using event handler

Data access module design

`Order`. A DAO must be written in a way that ensures that this situation is handled correctly. It must not allow one update to overwrite another. If a DAO implements updates by reading a record and then writing the updated record, it must use either pessimistic or optimistic locking. In the next section you'll see an example of a DAO that handles concurrent updates by updating database records without reading them first.

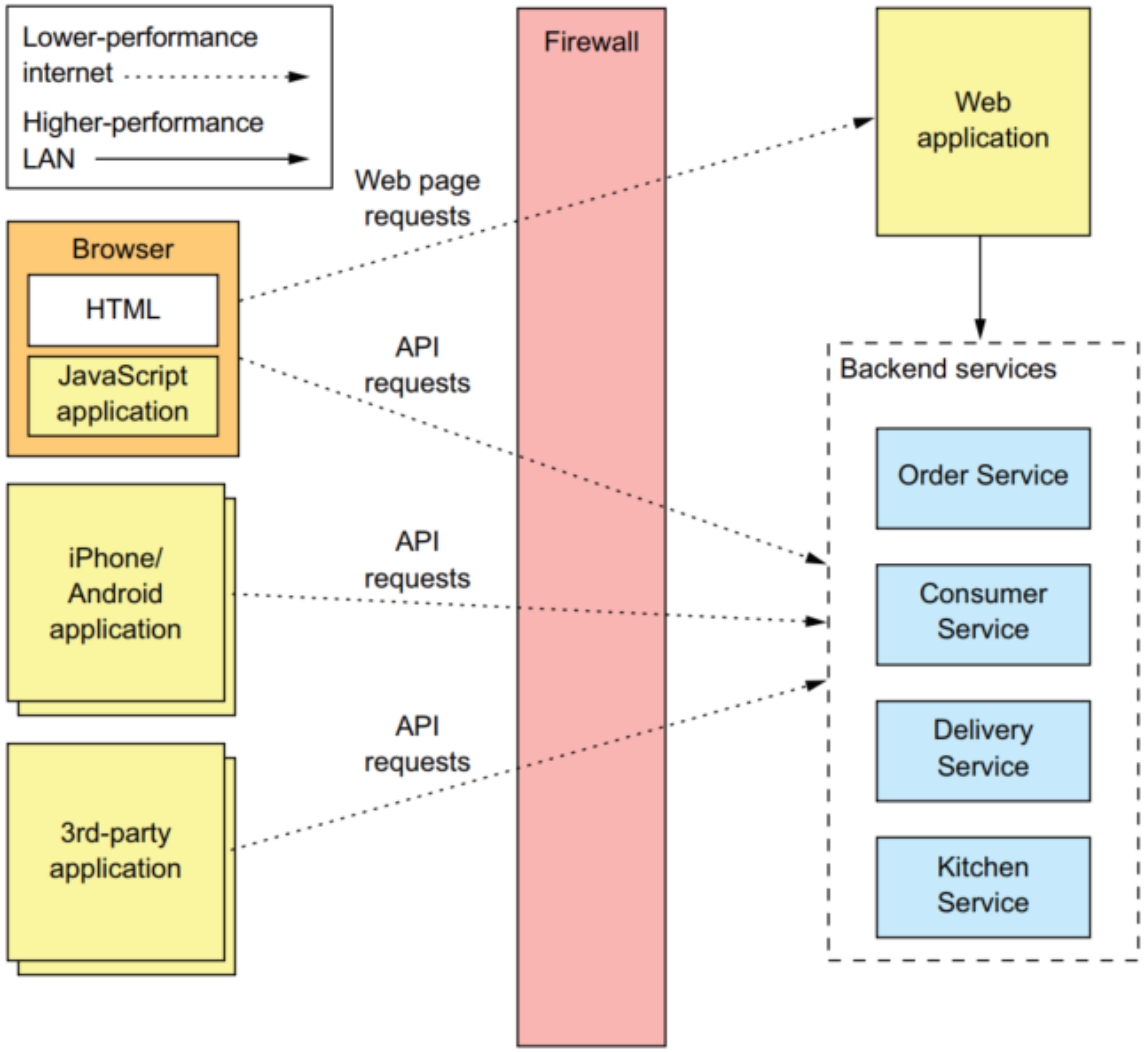
DETECTING DUPLICATE EVENTS

Using conditional update:

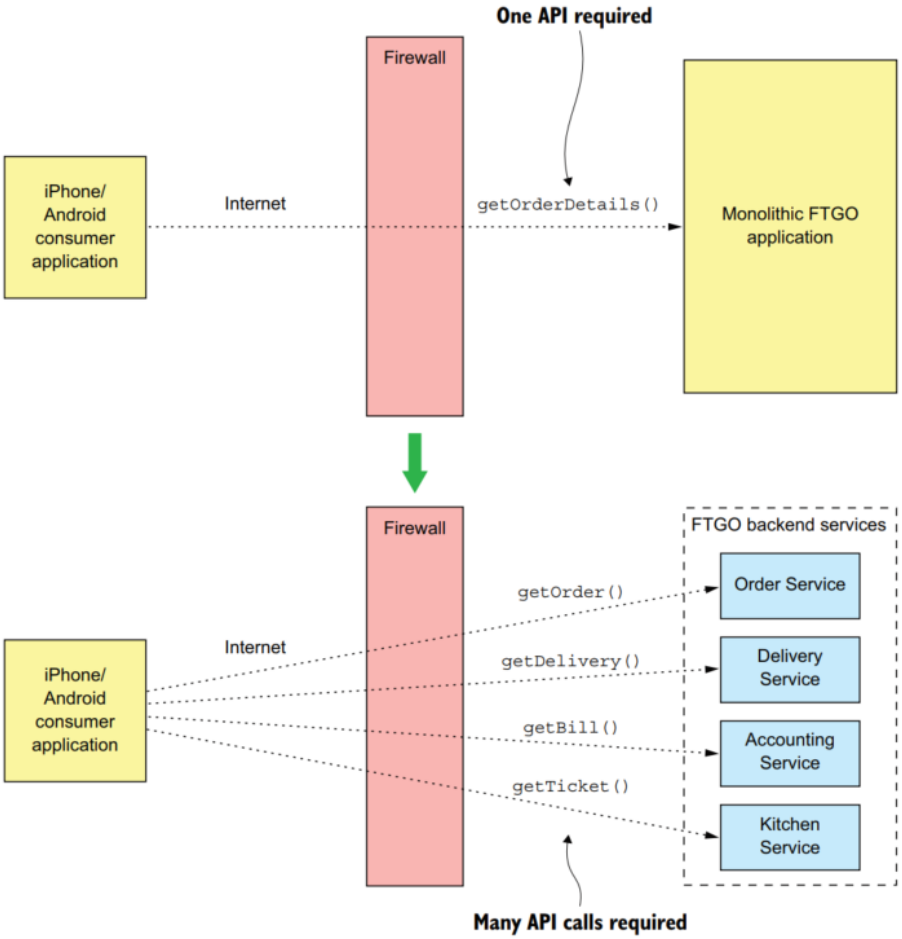
```
attribute_not_exists(«aggregateType»«aggregateId»)
OR «aggregateType»«aggregateId» < :eventId
```

The *condition expression* only allows the update if the attribute doesn't exist or the `eventId` is greater than the last processed event ID.

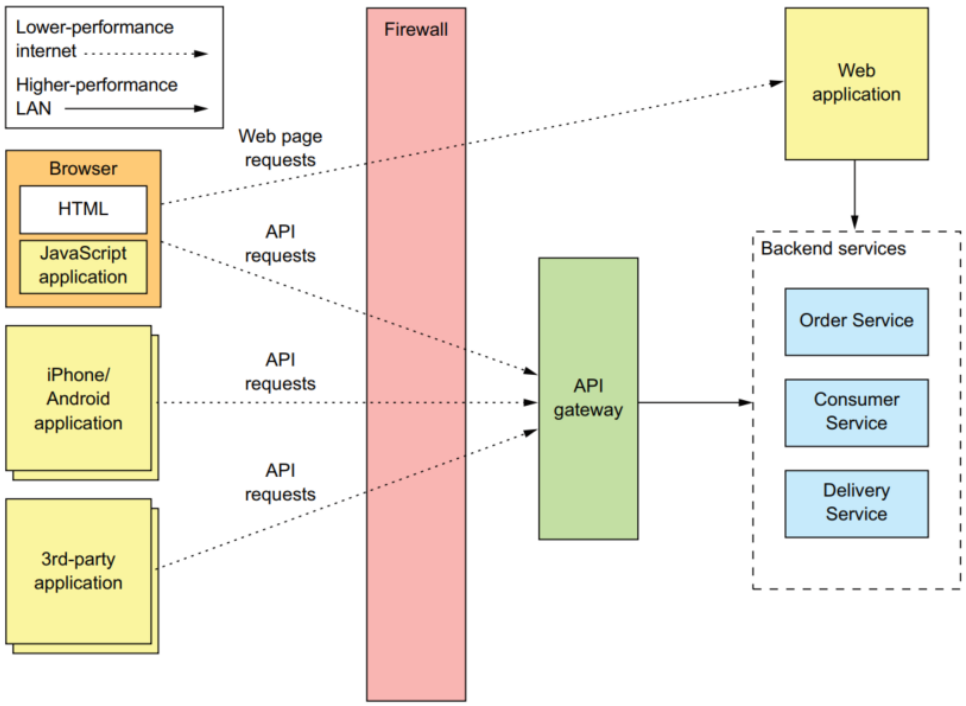
## 2.7 External API patterns




### 2.7.1 API design issues for the FTGO mobile client

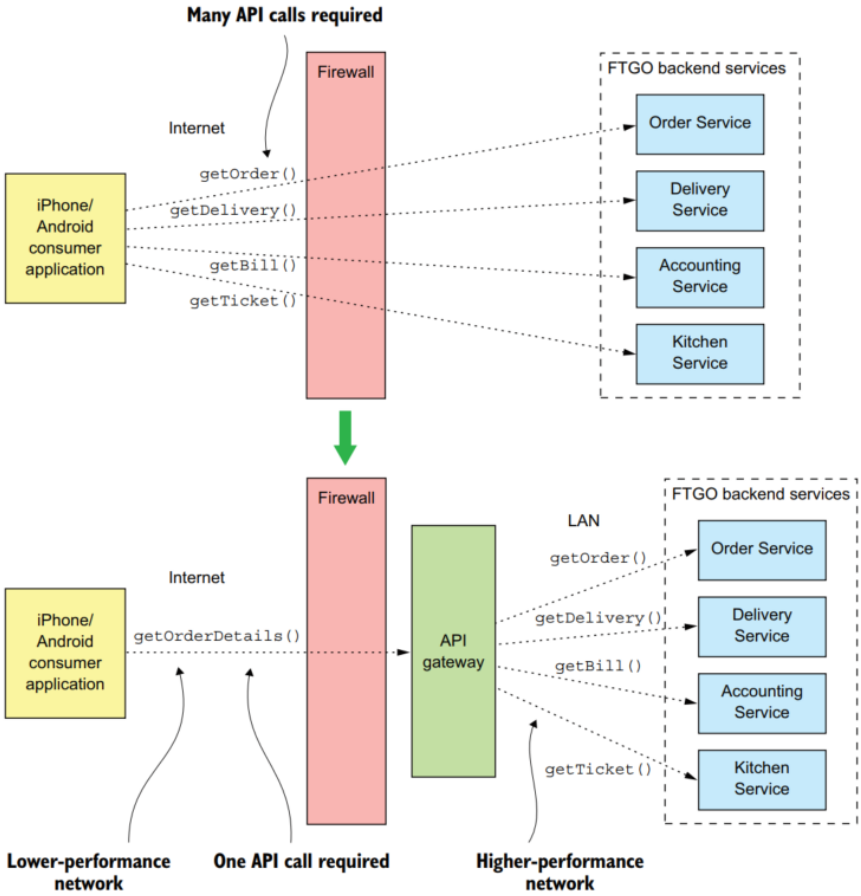



### 2.7.2 The API gateway pattern




- REQUEST ROUTING

- API COMPOSITION



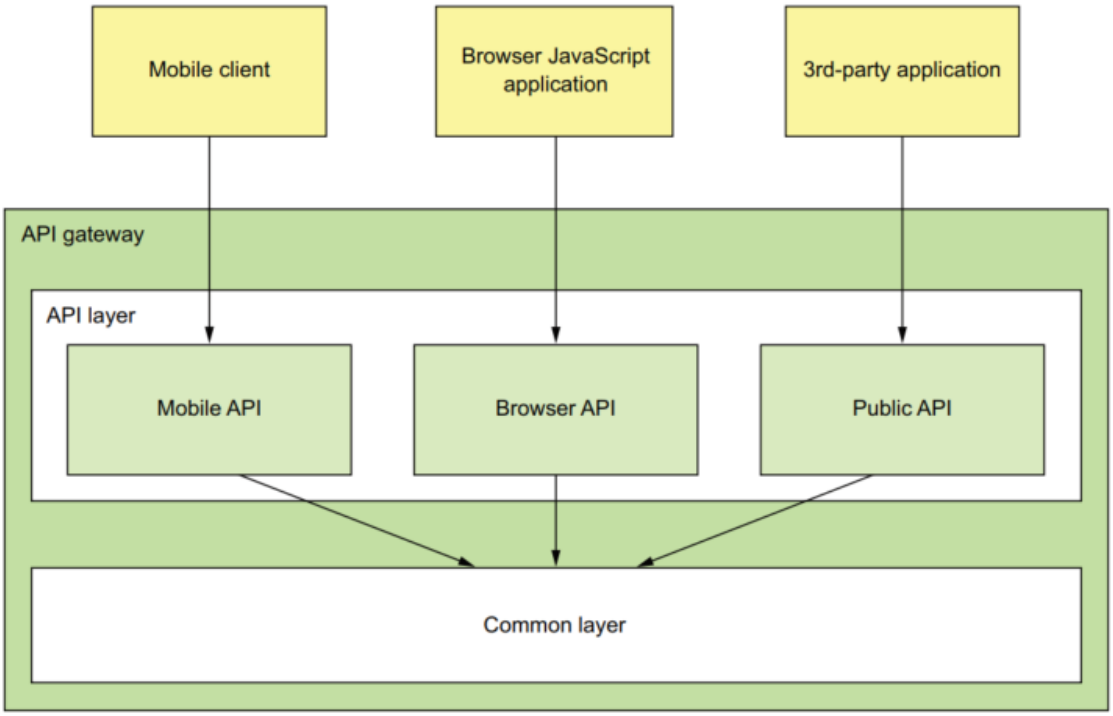

- PROTOCOL TRANSLATION

- IMPLEMENTING EDGE FUNCTIONS

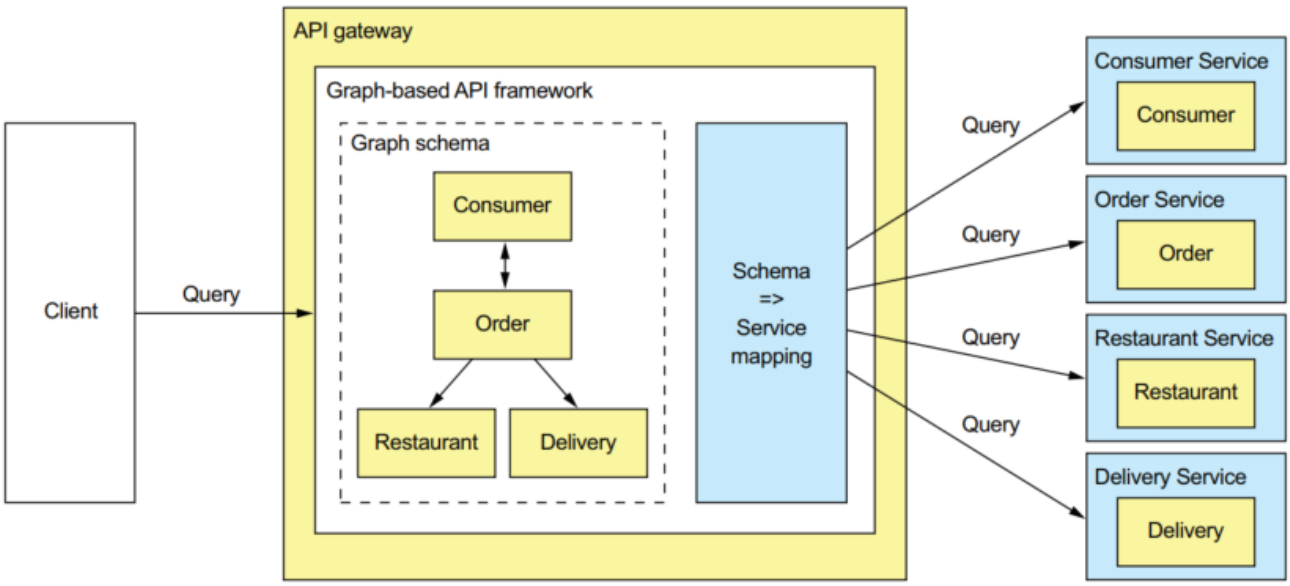
  - *Authentication*—Verifying the identity of the client making the request.
  - *Authorization*—Verifying that the client is authorized to perform that particular operation.
  - *Rate limiting*—Limiting how many requests per second from either a specific client and/or from all clients.
  - *Caching*—Cache responses to reduce the number of requests made to the services.
  - *Metrics collection*—Collect metrics on API usage for billing analytics purposes.
  - *Request logging*—Log requests.

### API GATEWAY ARCHITECTURE

An API gateway has a layered, modular architecture. Its architecture, shown in figure 8.5, consists of two layers: the API layer and a common layer. The API layer consists of one or more independent API modules. Each API module implements an API for a



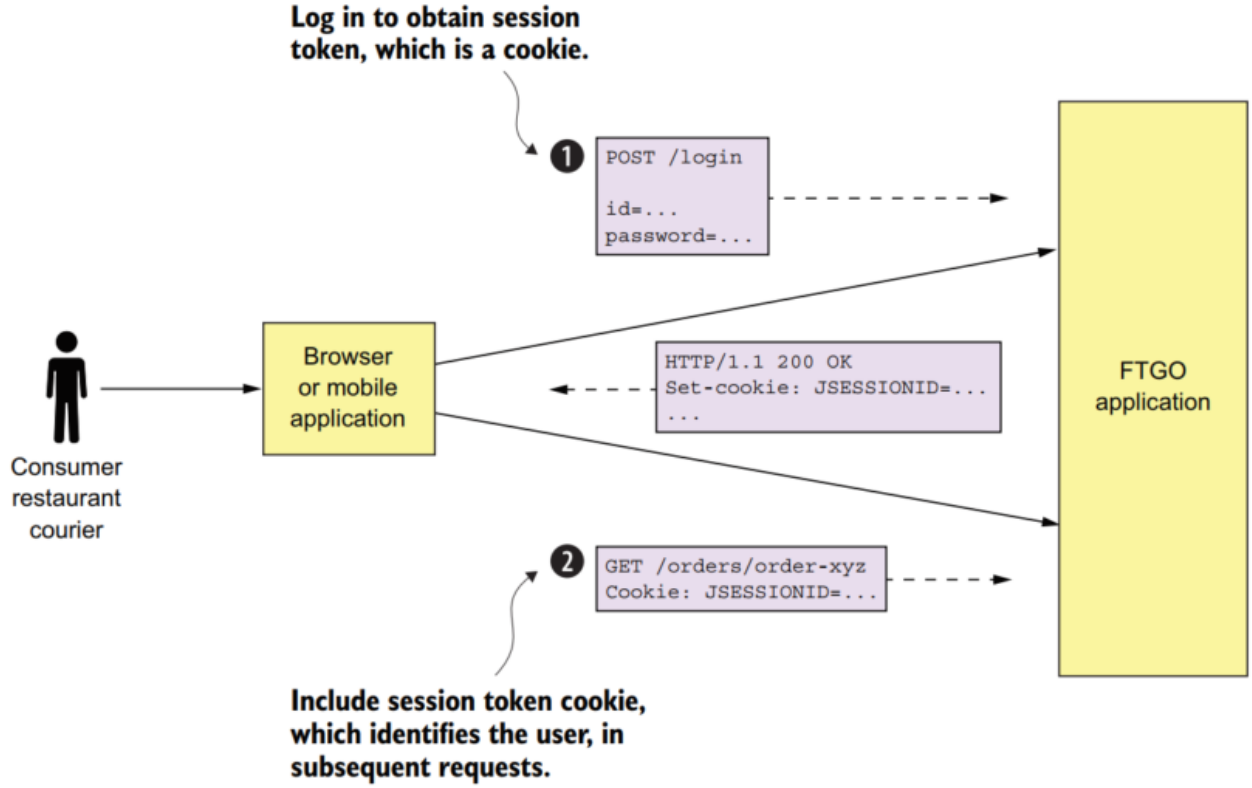

Implementing an API gateway using GraphQL

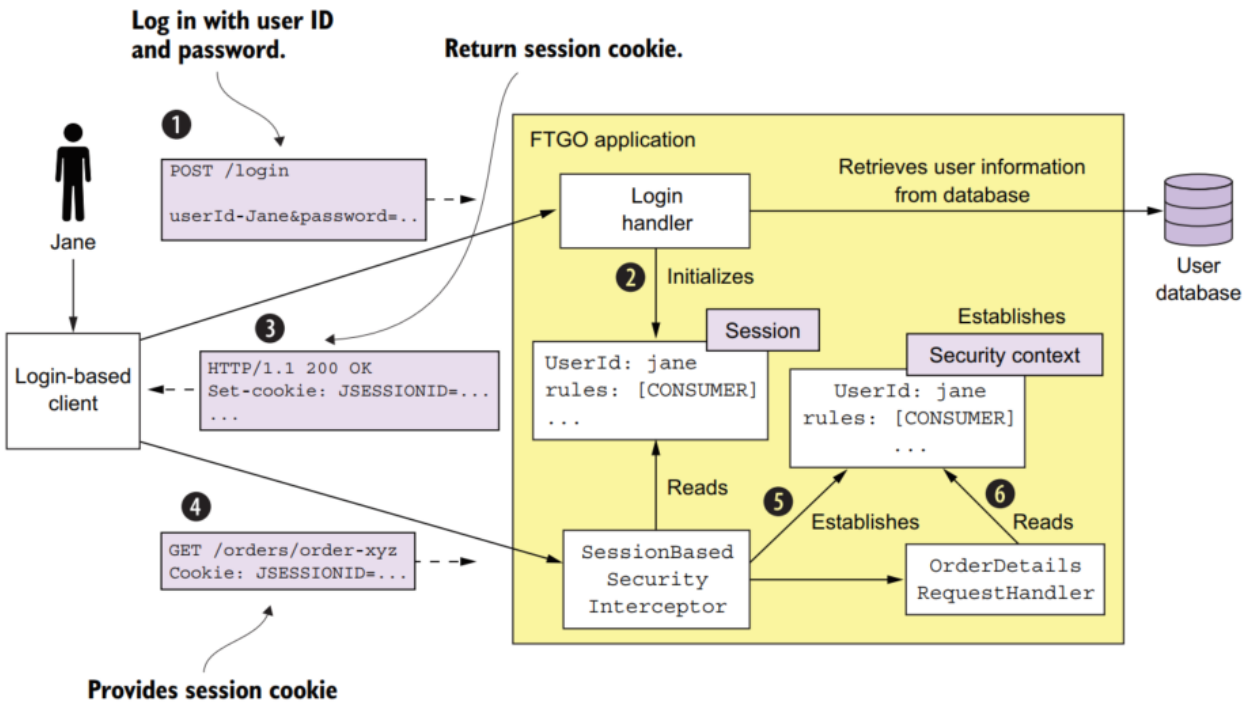



## 2.8 Developing production-ready services

### 2.8.1 Developing secure services

## Overview of security in a traditional monolithic application




## Implementing security in a microservice architecture

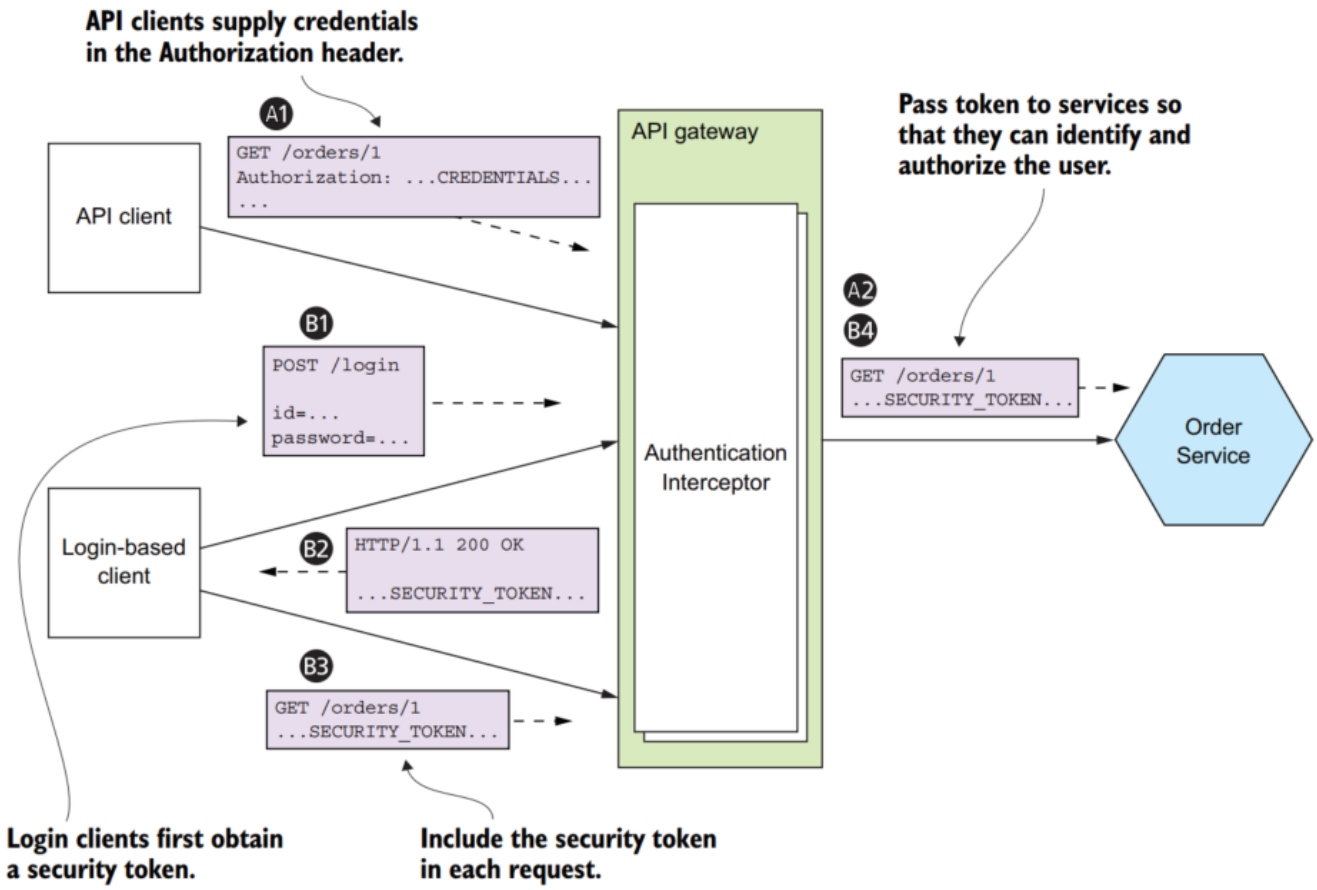
- API Gateway handle security
- Service also handle some aspect of security, for example **Order service** only see their own order.

## HANDLING AUTHENTICATION IN THE API GATEWAY

Another problem with implementing authentication in the services is that different clients authenticate in different ways. Pure API clients supply credentials with each request using, for example, basic authentication. Other clients might first log in and then supply a session token with each request. We want to avoid requiring services to handle a diverse set of authentication mechanisms.

A better approach is for the API gateway to authenticate a request before forwarding it to the services. Centralizing API authentication in the API gateway has the advantage that there's only one place to get right. As a result, there's a much smaller chance of a security vulnerability. Another benefit is that only the API gateway has to deal with the various different authentication mechanisms. It hides this complexity from the services.

- API clients include credentials in each request. Login-based clients POST the user's credentials to the API gateway's authentication and receive a session token.
- Once the API gateway has authenticated a request, it invokes one or more services



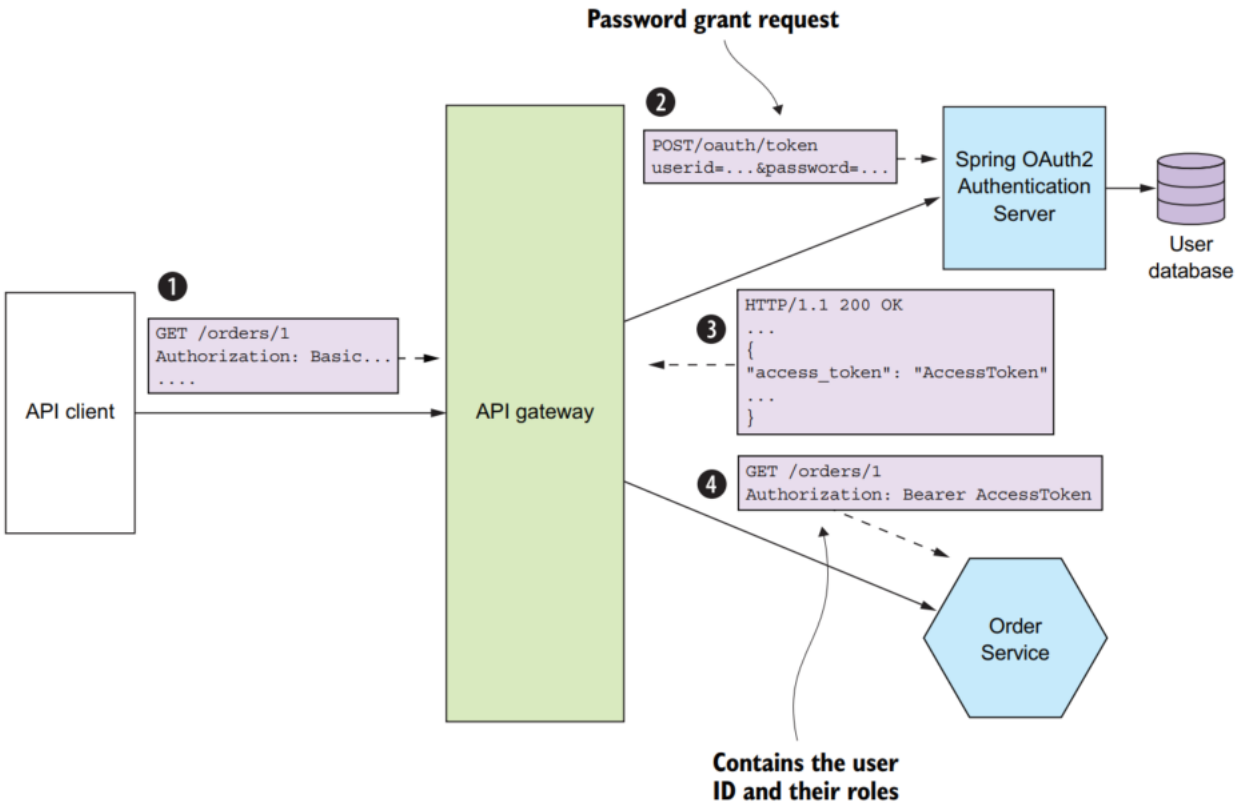

## HANDLING AUTHORIZATION

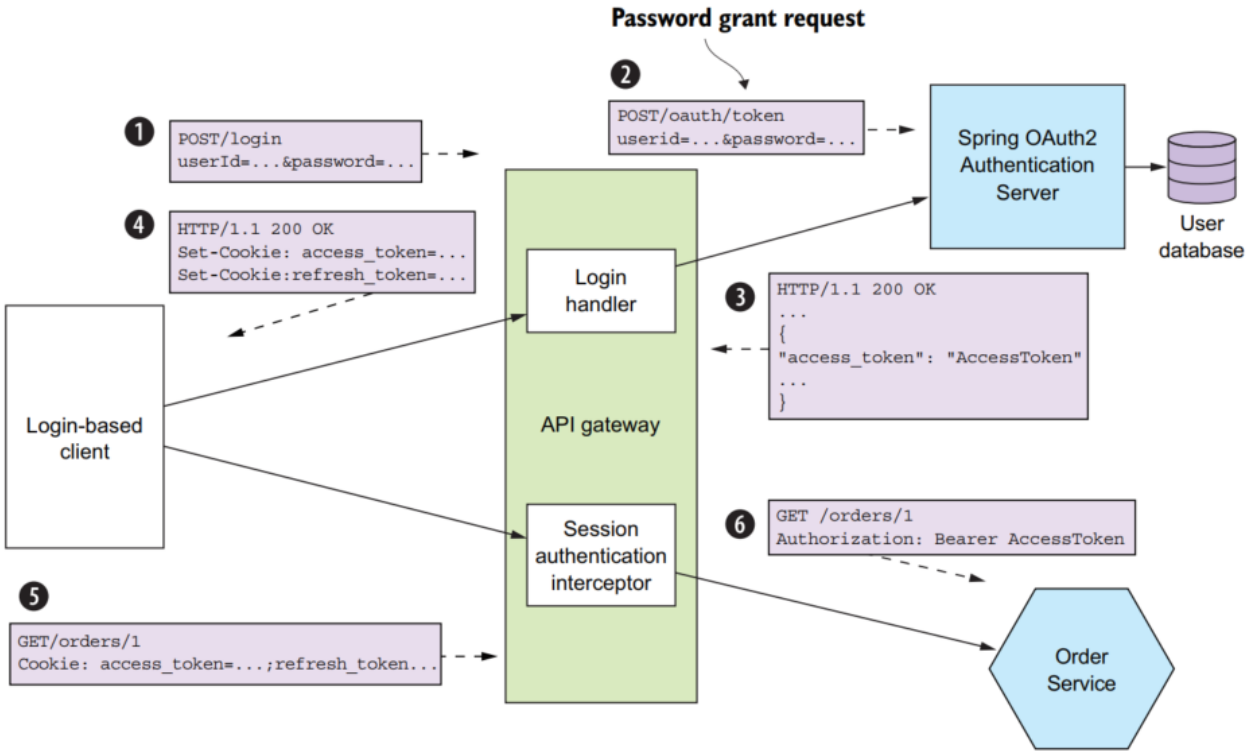
- Handle authorization in API gateway: decouple with service
- Handle authorization in service is better solution

## USING JWTS TO PASS USER IDENTITY AND ROLES

An alternative approach, which eliminates the call to the security service, is to use a *transparent* token containing information about the user. One such popular standard for transparent tokens is the JSON Web Token (JWT). JWT is standard way to securely represent claims, such as user identity and roles, between two parties. A JWT has a payload, which is a JSON object that contains information about the user, such as their identity and roles, and other metadata, such as an expiration date. It's signed with a secret that's only known to the creator of the JWT, such as the API gateway and the recipient of the JWT, such as a service. The secret ensures that a malicious third party can't forge or tamper with a JWT.

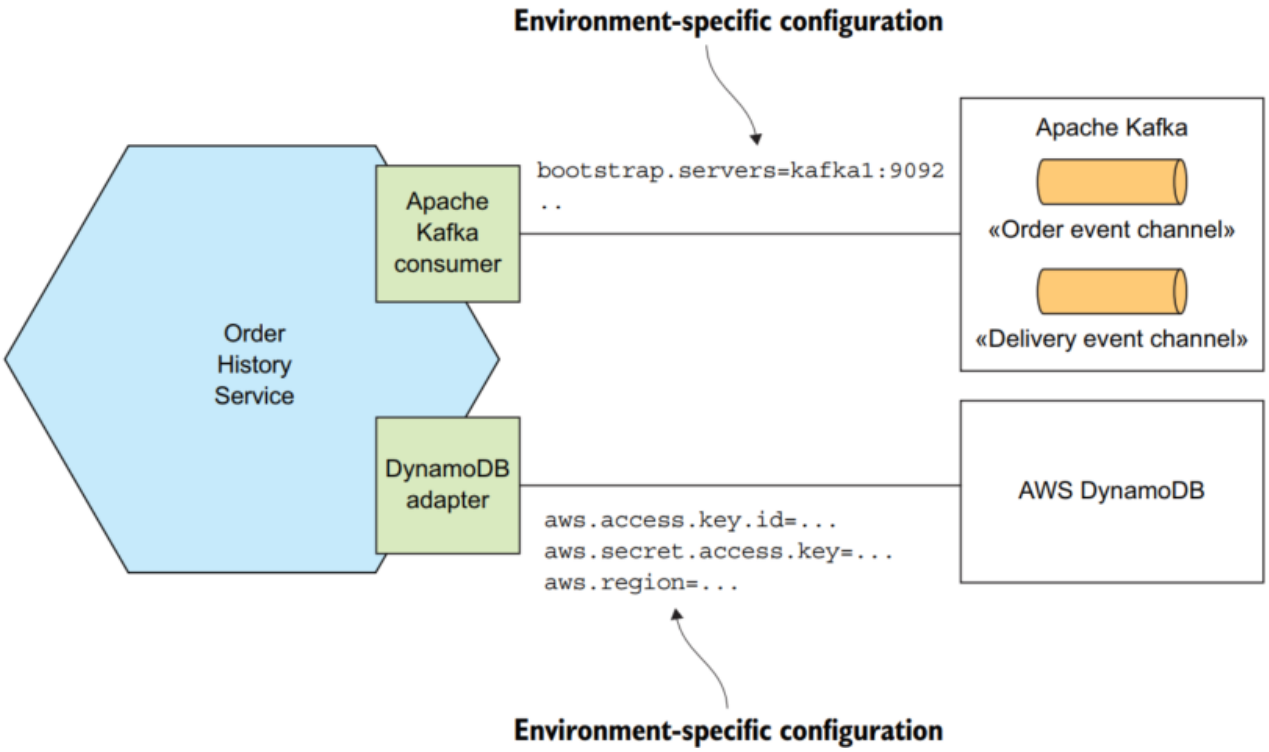
## USING OAUTH 2.0 IN A MICROSERVICE ARCHITECTURE






### 2.8.2 Configuration Service

Designing configurable services

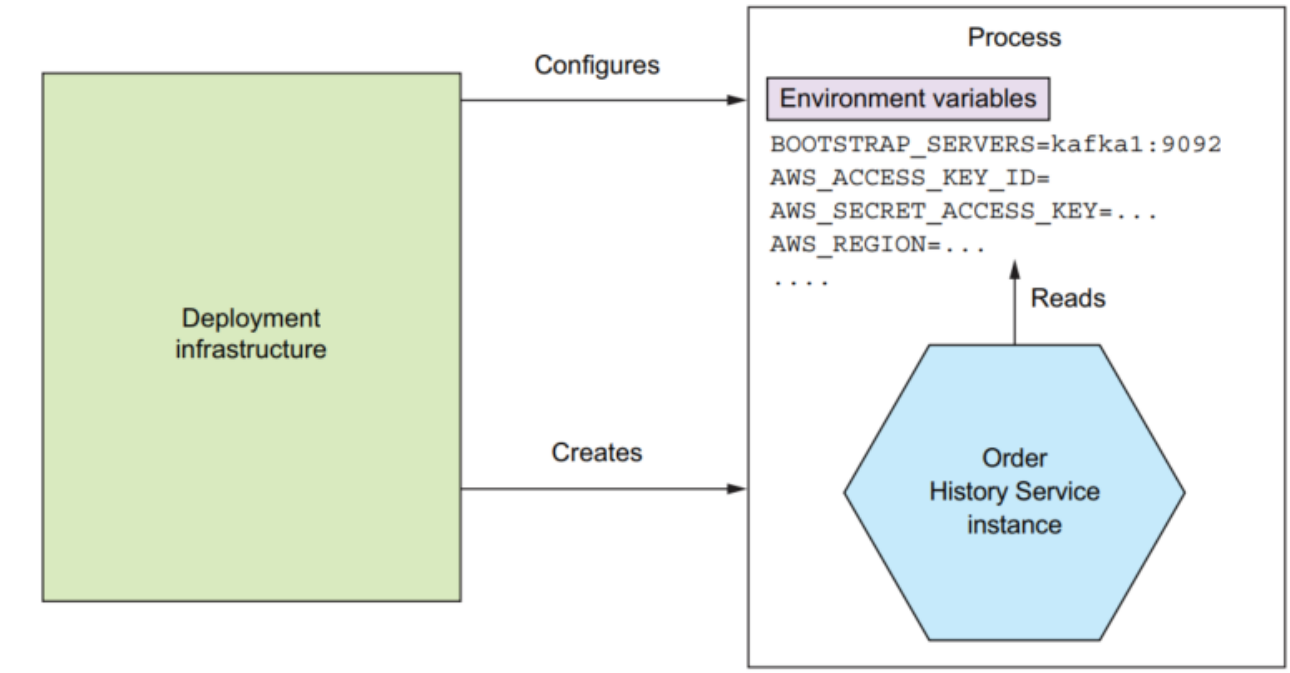


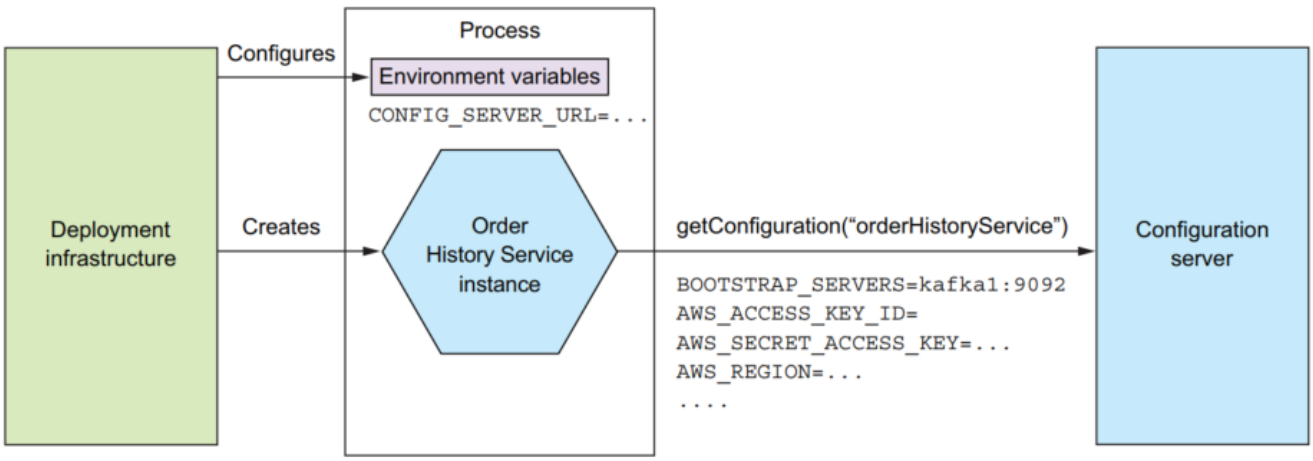

Configuration store:

- Hashicorp

- AWS Parameter store

### Using push-based externalized configuration

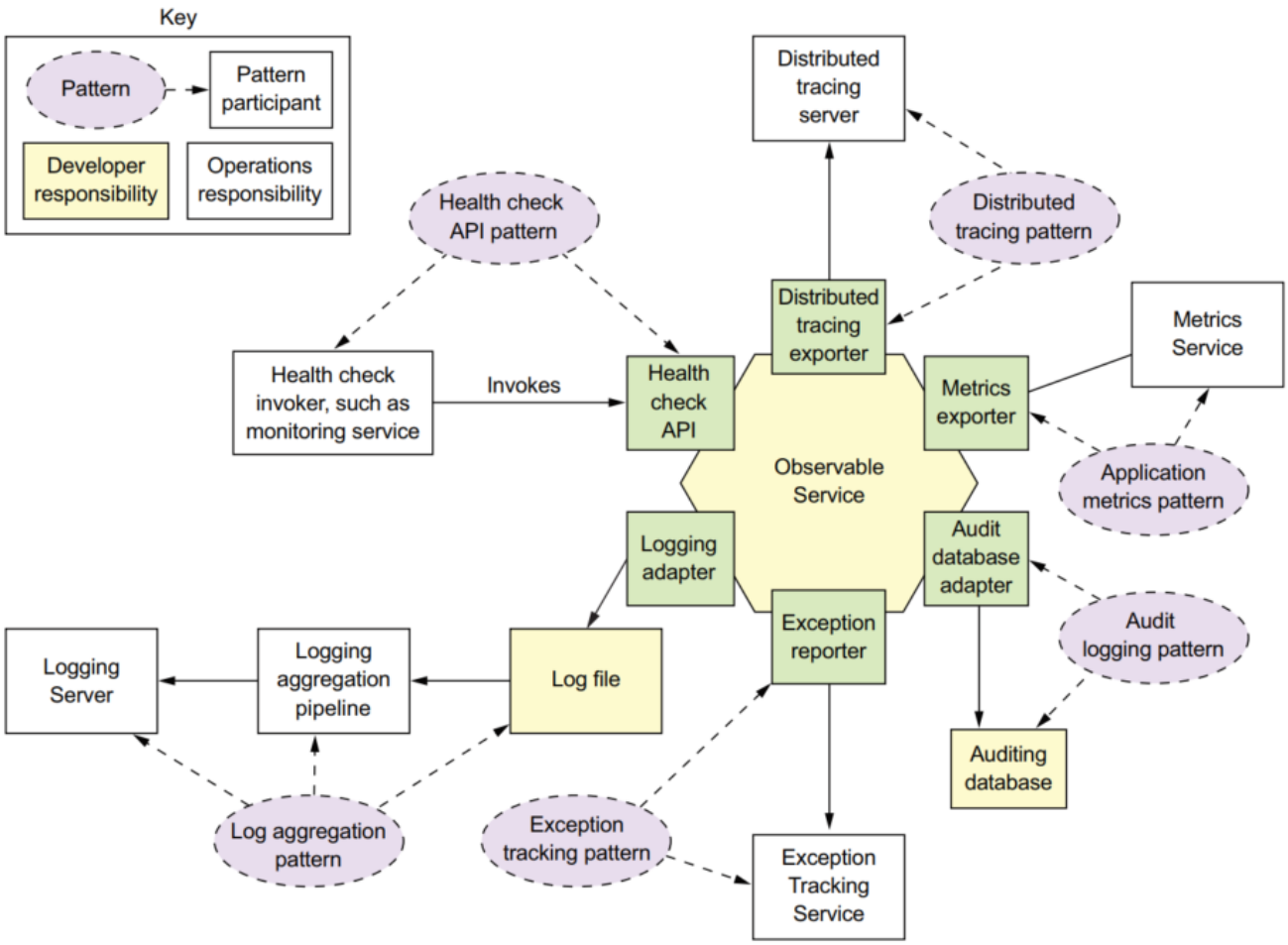



### 2.8.3 Designing observable services

Observable services:

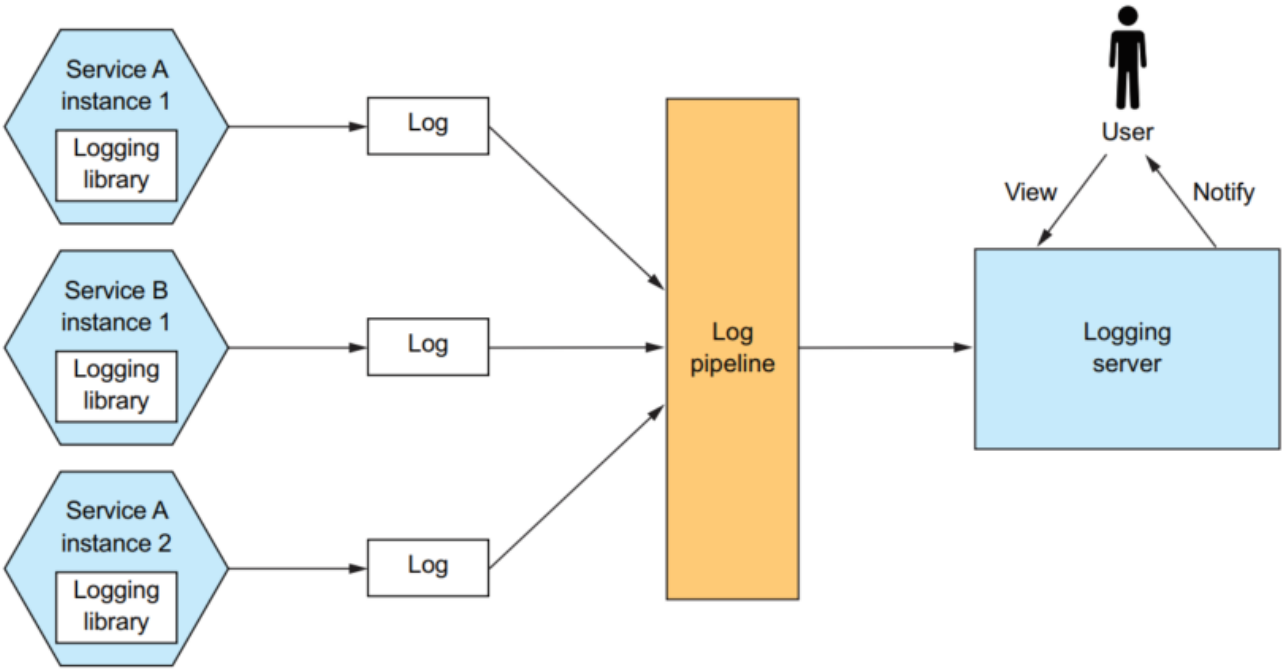
- Health Check API
- Log aggregation




#### INVOKING THE HEALTH CHECK ENDPOINT

A health check endpoint isn't much use if nobody calls it. When you deploy your service, you must configure the deployment infrastructure to invoke the endpoint. How you do that depends on the specific details of your deployment infrastructure. For example, as described in chapter 3, you can configure some service registries, such as Netflix Eureka, to invoke the health check endpoint in order to determine whether traffic should be routed to the service instance. Chapter 12 discusses how to configure Docker and Kubernetes to invoke a health check endpoint.

### Applying the Log aggregation pattern

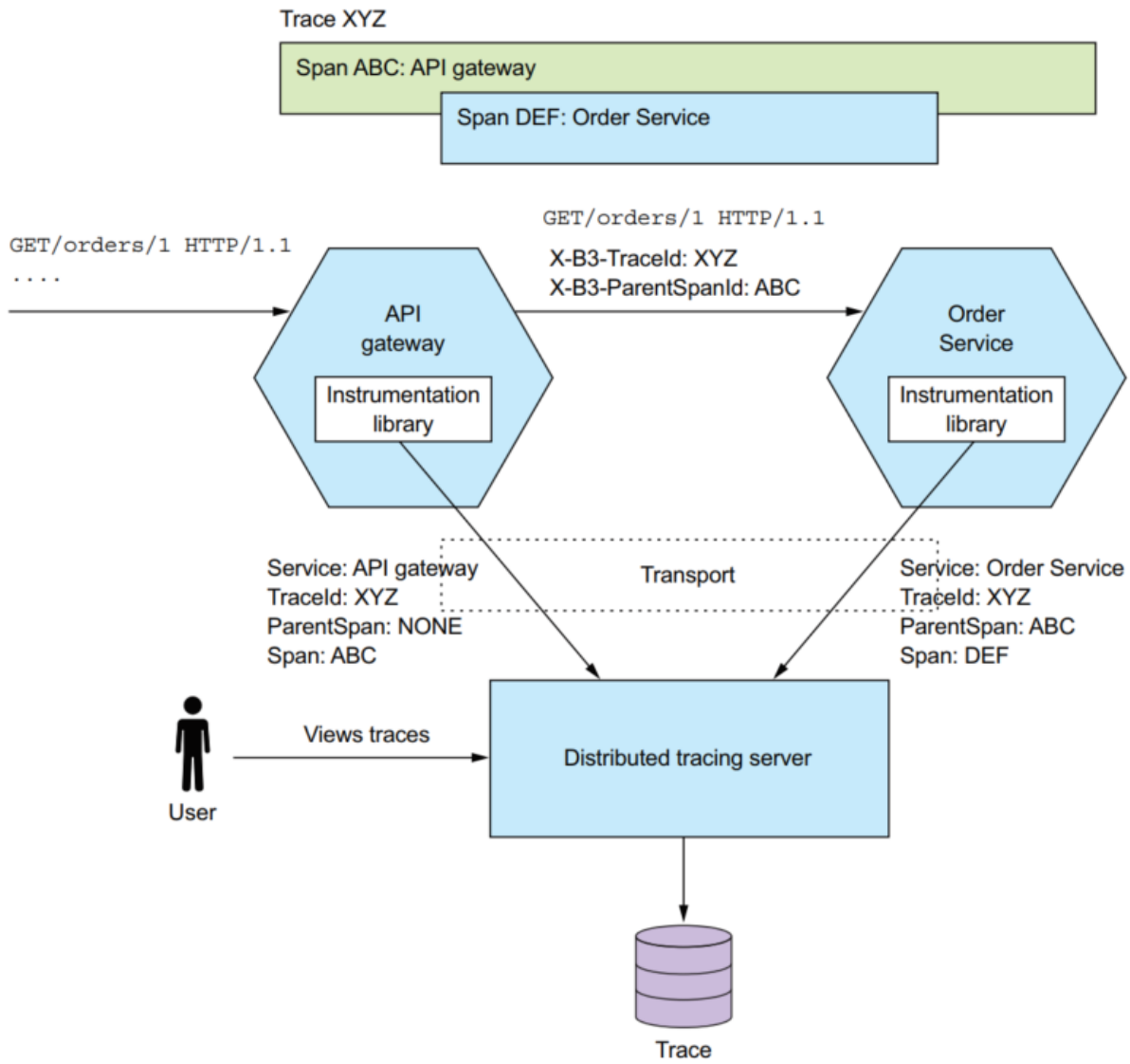


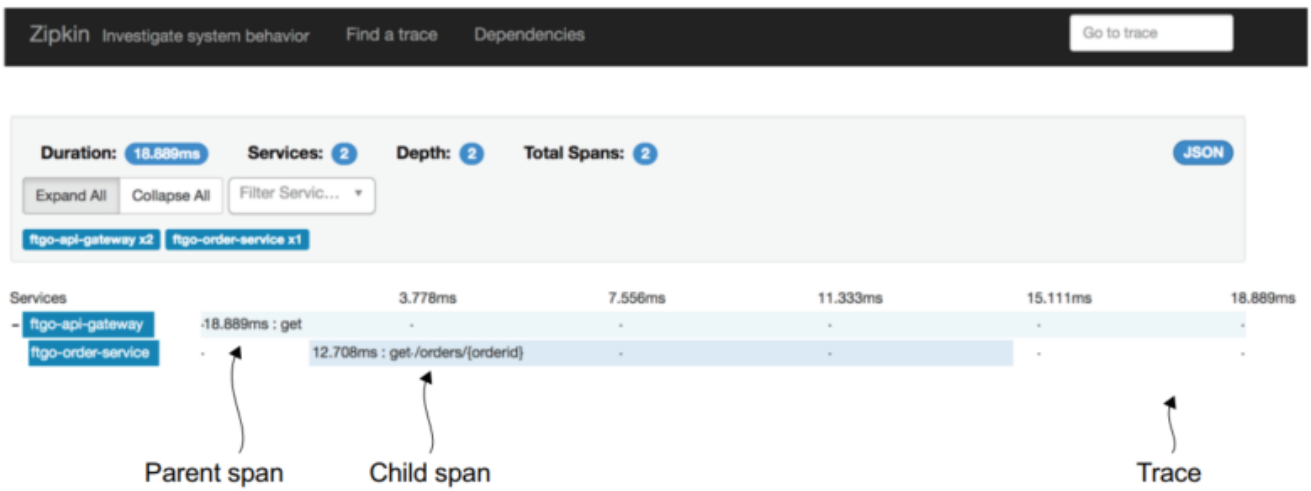

### THE LOG AGGREGATION INFRASTRUCTURE

The logging infrastructure is responsible for aggregating the logs, storing them, and enabling the user to search them. One popular logging infrastructure is the ELK stack. ELK consists of three open source products:

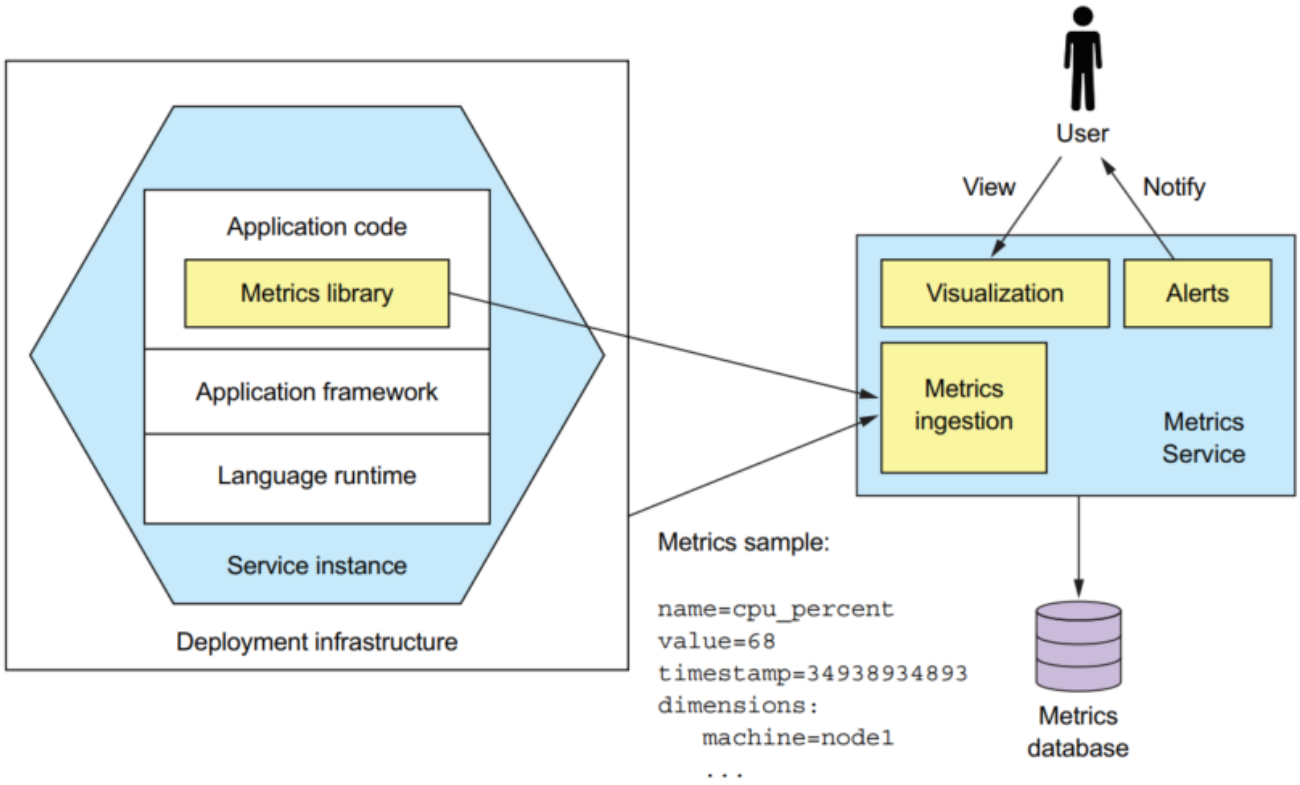
- *Elasticsearch*—A text search-oriented NoSQL database that's used as the logging server
- *Logstash*—A log pipeline that aggregates the service logs and writes them to Elasticsearch
- *Kibana*—A visualization tool for Elasticsearch

## Using the Distributed tracing pattern

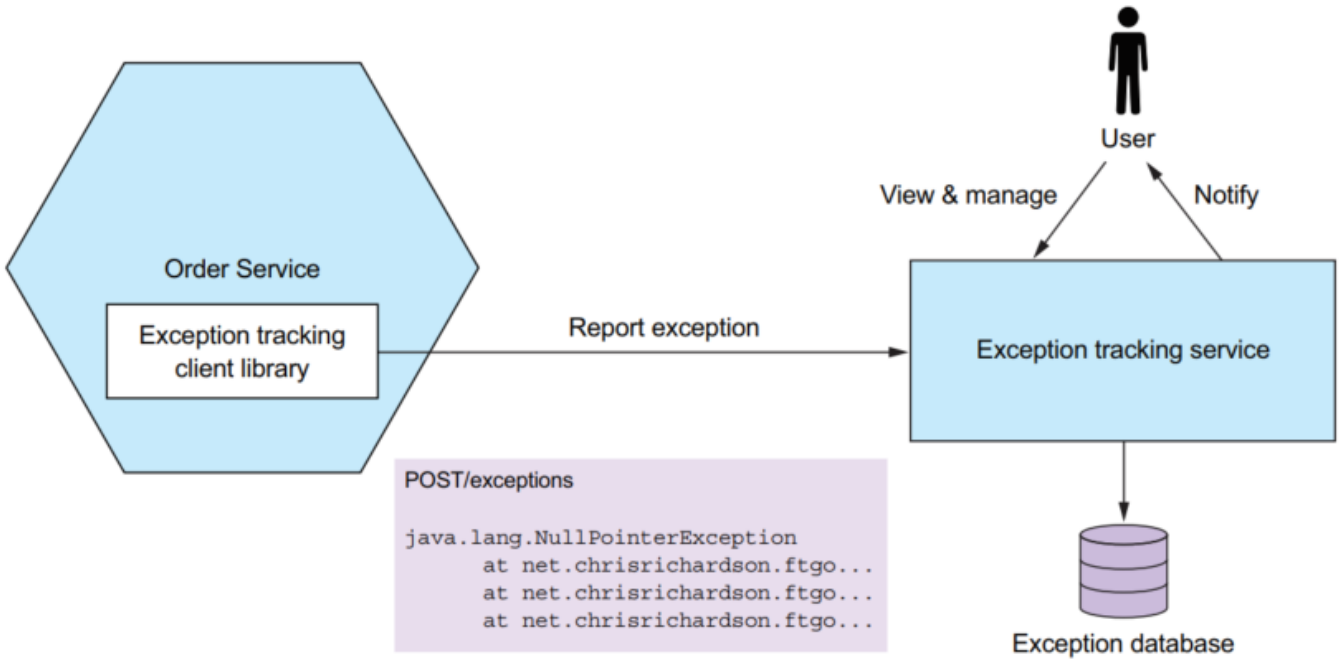


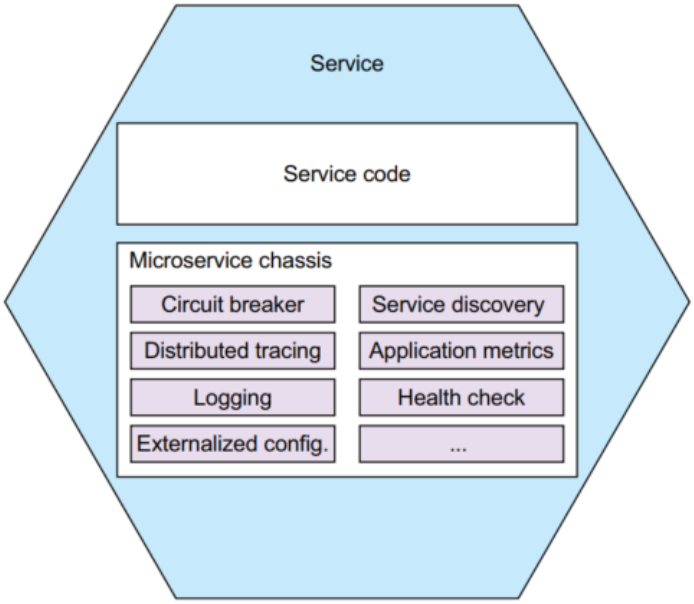

## Applying the Application metrics pattern



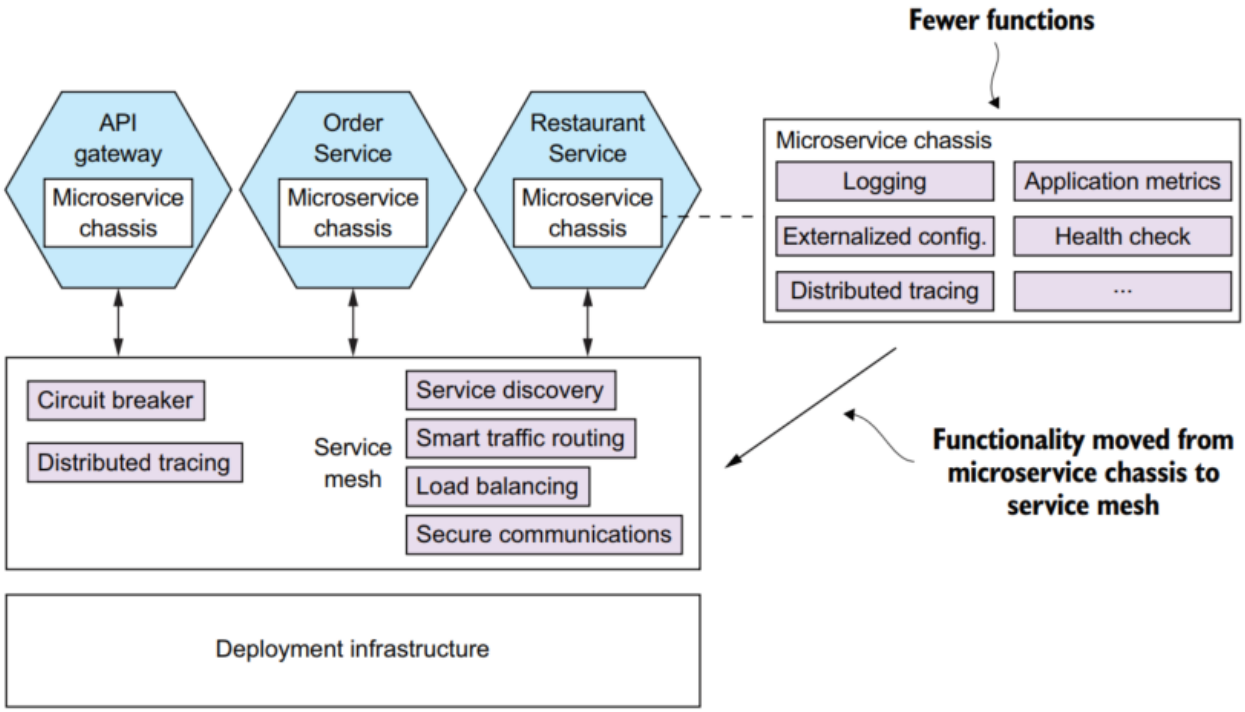

Using the Exception tracking pattern




### 2.8.4 Chassis pattern




## From microservice chassis to service mesh



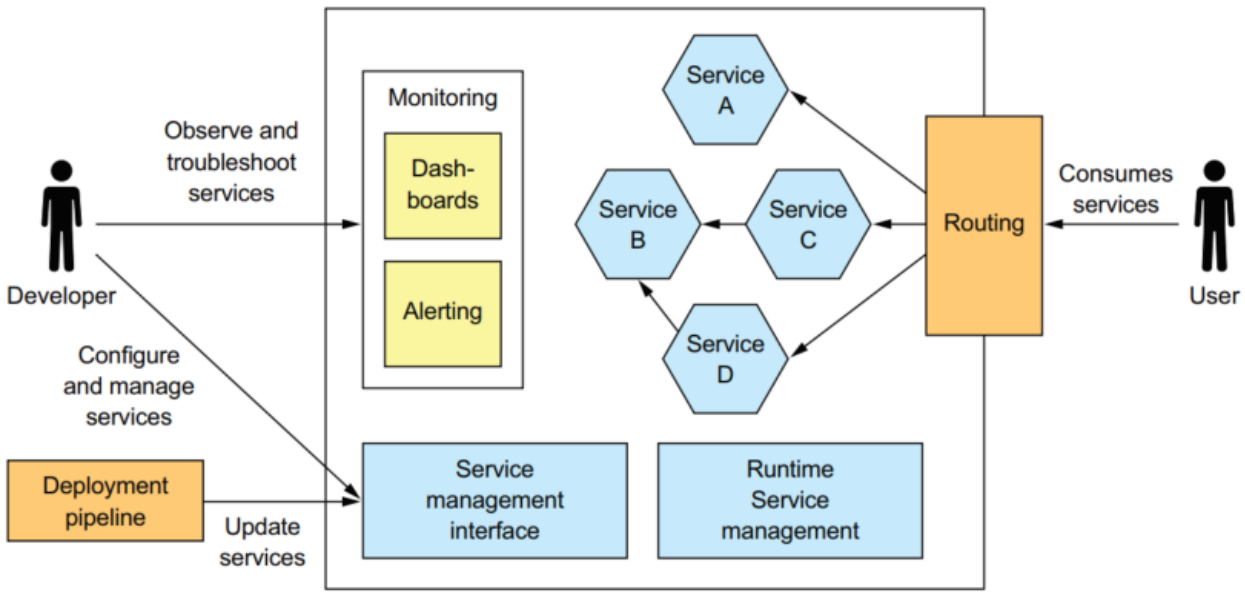
**The current state of service mesh implementations**

There are various service mesh implementations, including the following:

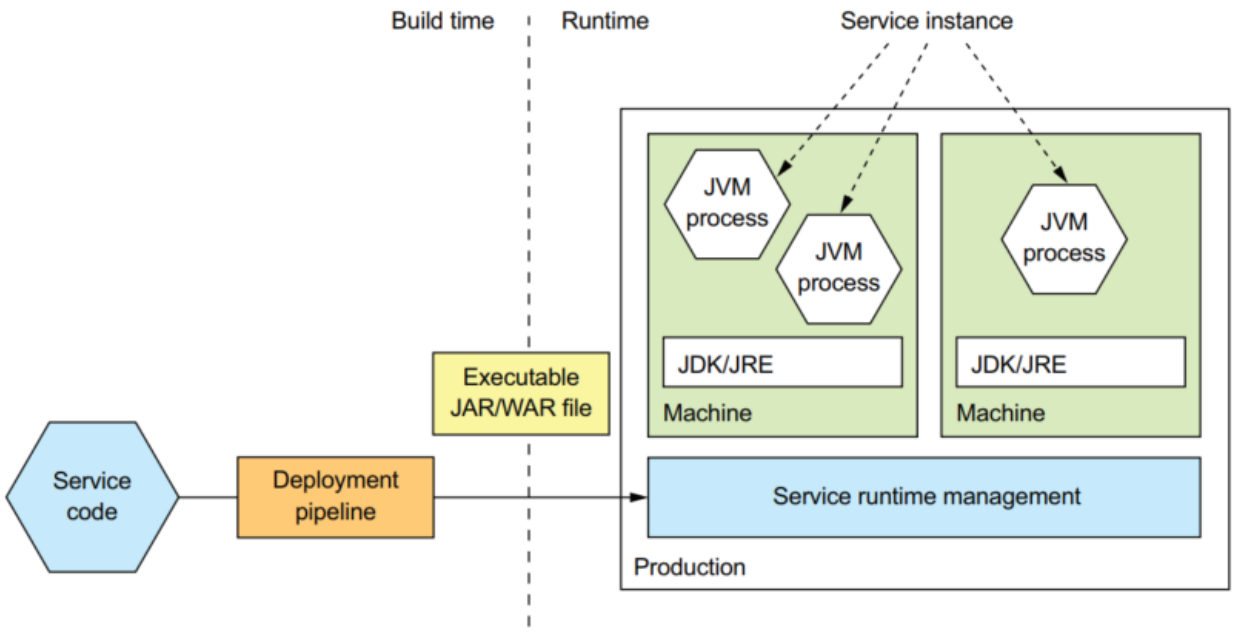
- Istio (https://istio.io)
- Linkerd (https://linkerd.io)
- Conduit (https://conduit.io)

As of the time of writing, Linkerd is the most mature, with Istio and Conduit still under active development. For more information about this exciting new technology, take a look at each product's documentation.

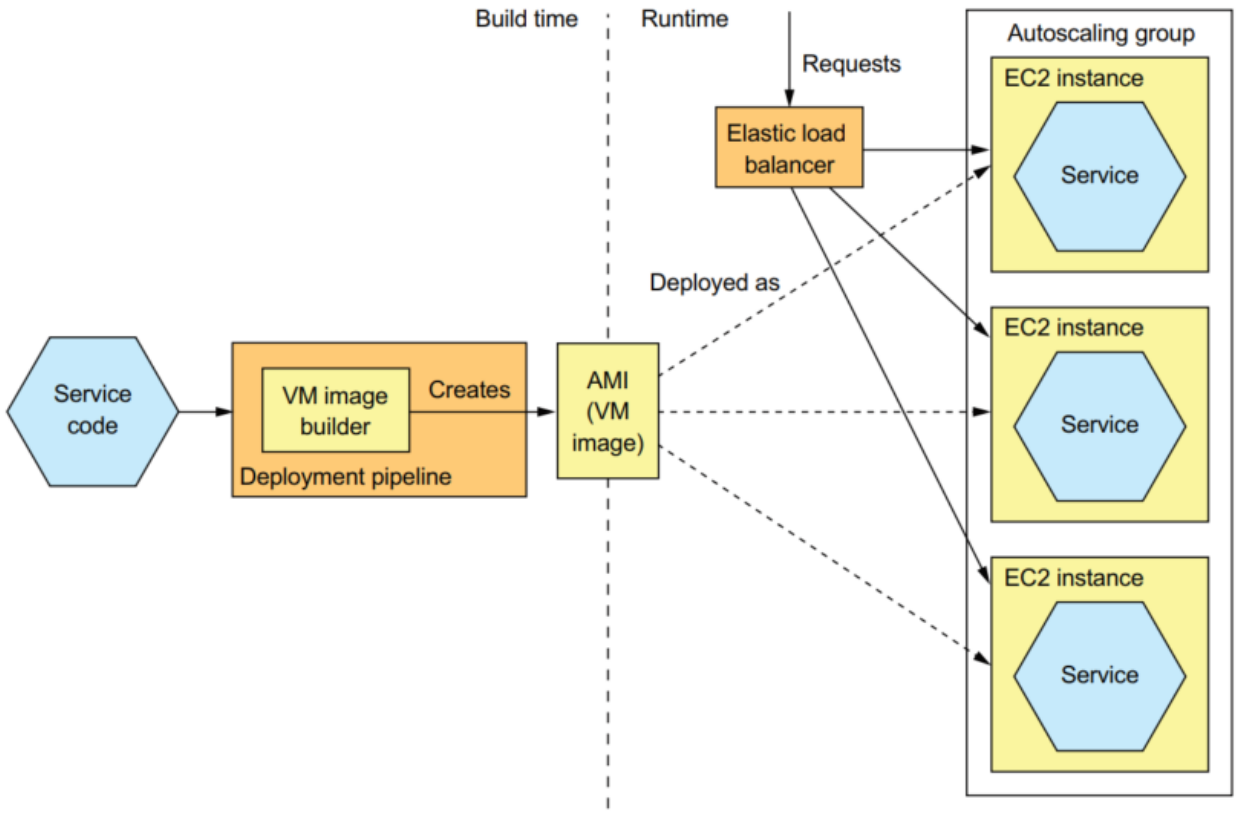
## 2.9 Deploying microservices




### 2.9.1 Deploying services using the Language-specific

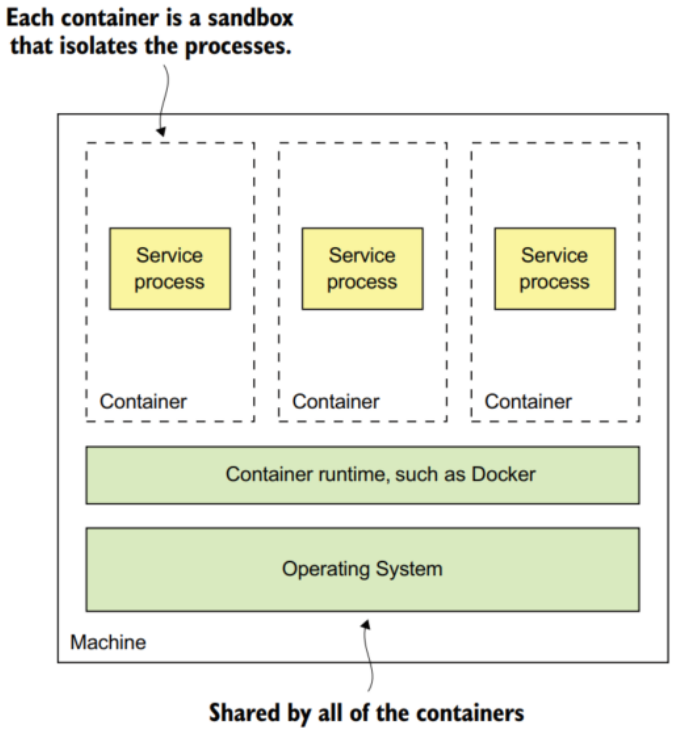



### 2.9.2 Deploying services using the Service as a virtual machine

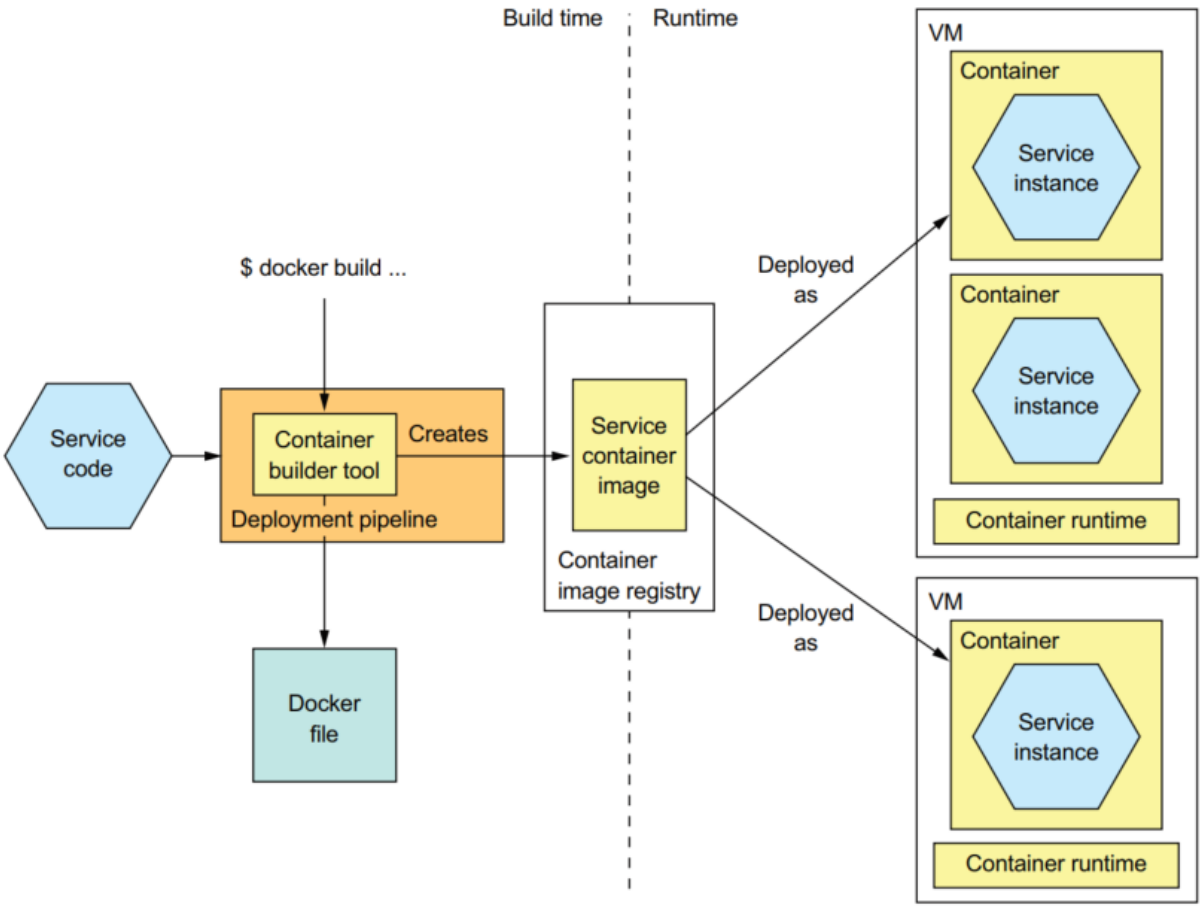



| PROS | CONS |
| --- | --- |
| | |

### 2.9.3 Deploying services using the Service as a container pattern




When you create a container, you can specify its CPU, memory resources, and, depending on the container implementation, perhaps the I/O resources. The container runtime enforces these limits and prevents a container from hogging the resources of its machine. When using a Docker orchestration framework such as Kubernetes, it's especially important to specify a container's resources. That's because the orchestration framework uses a container's requested resources to select the machine to run the container and thereby ensure that machines aren't overloaded.




| PROS | CONS |
| --- | --- |
| Encapsulation of the technology stack in which the API for managing your services becomes the container API | One significant drawback of containers is that you're responsible for the undifferentiated heavy lifting of administering the container images. You must patch the operating system and runtime. |
| Service instances are isolated. | |
| Service instances's resources are constrained | |

### 2.9.4 Deploying the FTGO application with Kubernetes

A Docker orchestration framework, such as Kubernetes , has three main functions:

- *Resource management*—Treats a cluster of machines as a pool of CPU, memory, and storage volumes, turning the collection of machines into a single machine.
- *Scheduling*—Selects the machine to run your container. By default, scheduling considers the resource requirements of the container and each node's available resources. It might also implement *affinity*, which colocates containers on the same node, and *anti-affinity*, which places containers on different nodes.
- *Service management*—Implements the concept of named and versioned services that map directly to services in the microservice architecture. The orchestration framework ensures that the desired number of healthy instances is running at all times. It load balances requests across them. The orchestration framework performs rolling upgrades of services and lets you roll back to an old version.

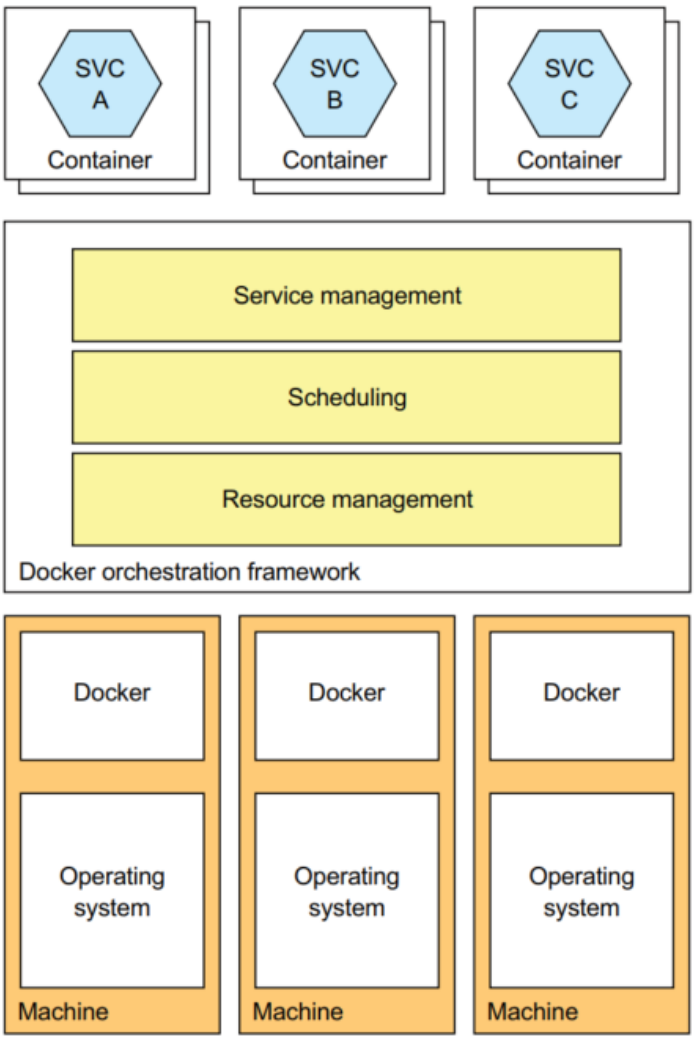
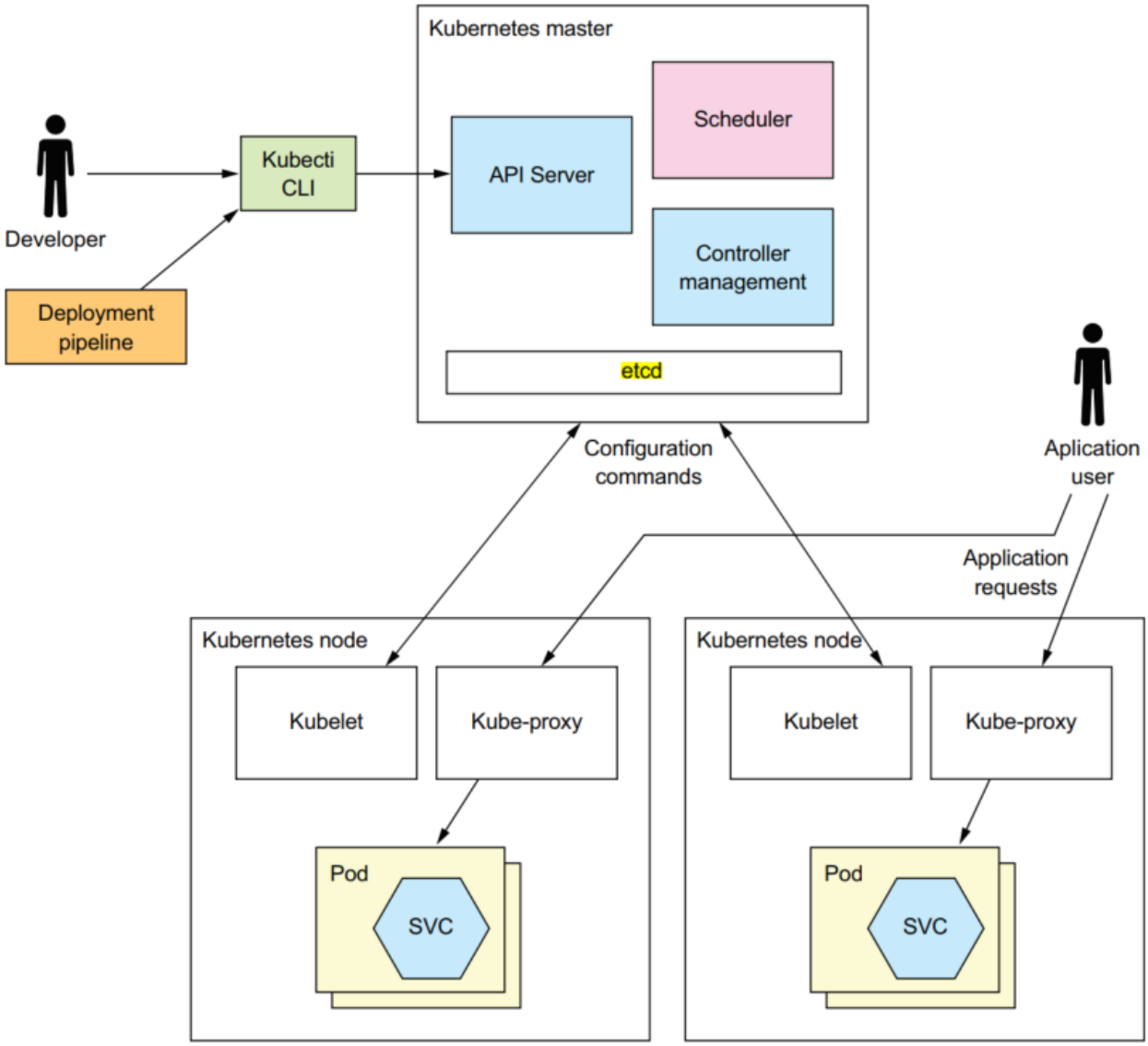



**Figure 12.9 A Docker orchestration framework turns a set of machines running Docker into a cluster of resources. It assigns containers to machines. The framework attempts to keep the desired number of healthy containers running at all times.**



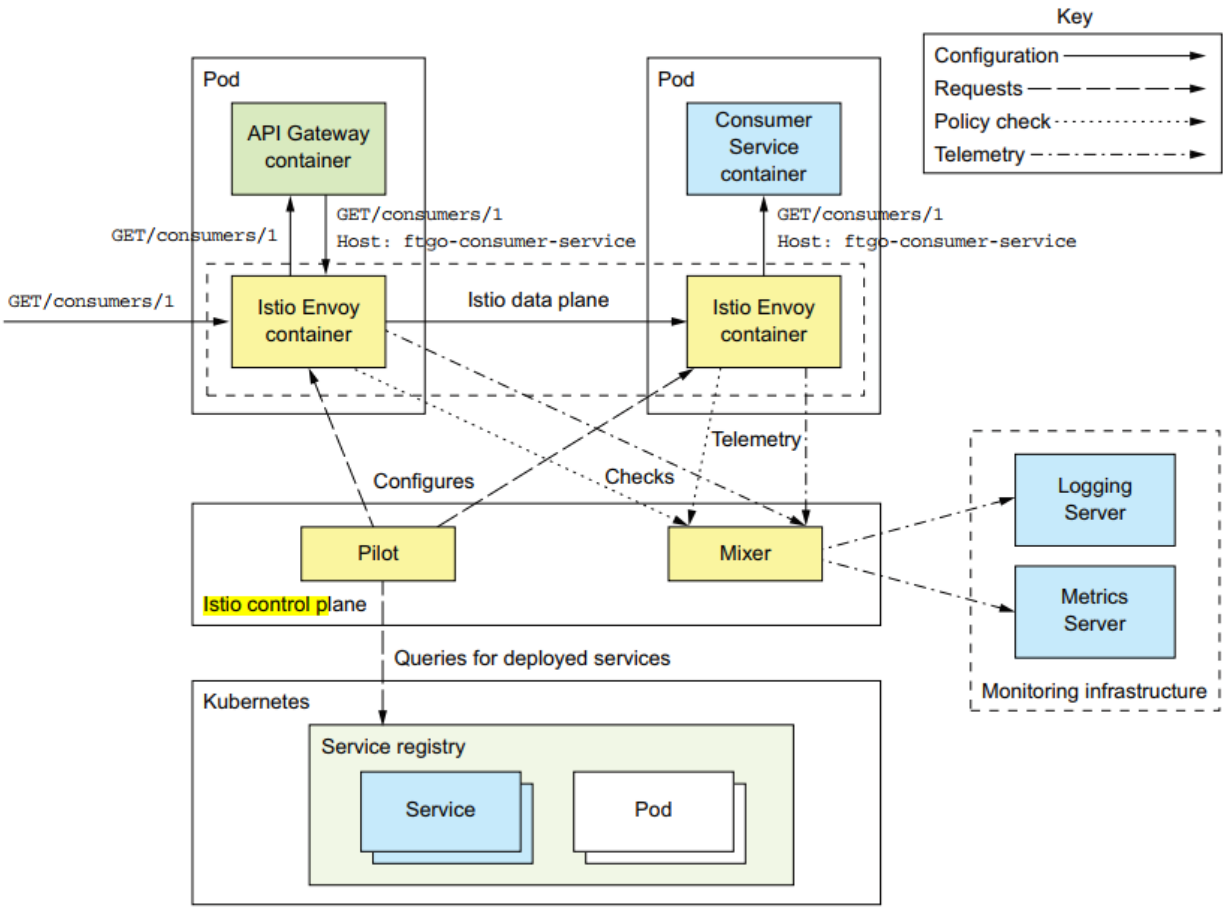

- K8s support rolling update
- Using a service mesh to separate deployment from release

You then deploy a service into production using the following steps:

1. Deploy the new version into production without routing any end-user requests to it.
2. Test it in production.
3. Release it to a small number of end users.
4. Incrementally release it to an increasingly larger number of users until it's handling all the production traffic.
5. If at any point there's an issue, revert back to the old version—otherwise, once you're confident the new version is working correctly, delete the old version.

## Istio function

- Load balancing
- Fine-grain control: rule how service A can talk to service B
- Access control
- Visibility: logging, graph, health check




**Pilot:** Support AB testing, fine grain control of service communication, retry, circuit breaker

**Citadel**: secure message, policy enfore

**Mixer**: Telemetry

### 2.9.5 Summary pros and cons

Container

| Pros | Cons |
| --- | --- |
| Service instances are isolated. | Manage container |
| No need to manage service | |
| Usage-based pricing | |

**AWS lambda:**

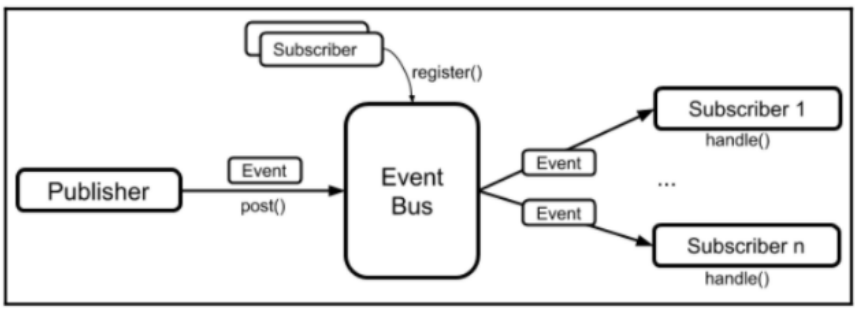
| Pros | Cons |
| --- | --- |
| Well integrate with AWS | Latency |
| No need to manage service | Limit event/request from third party |
| Usage-based pricing | |

## 3. Scale Architecture

- Event notification
- Event-carried state transfer
- Event sourcing
- CQRS

**Event notification**

This behavior is widely used by event bus libraries that allow publish-subscribe communication among the components that are part of an application. The most common use cases for these libraries are targeted towards the UI, but they are also applicable to other parts of the system in the backend. The following diagram demonstrates how an event is sent to the bus and then propagated to all subscribers, which were previously registered:

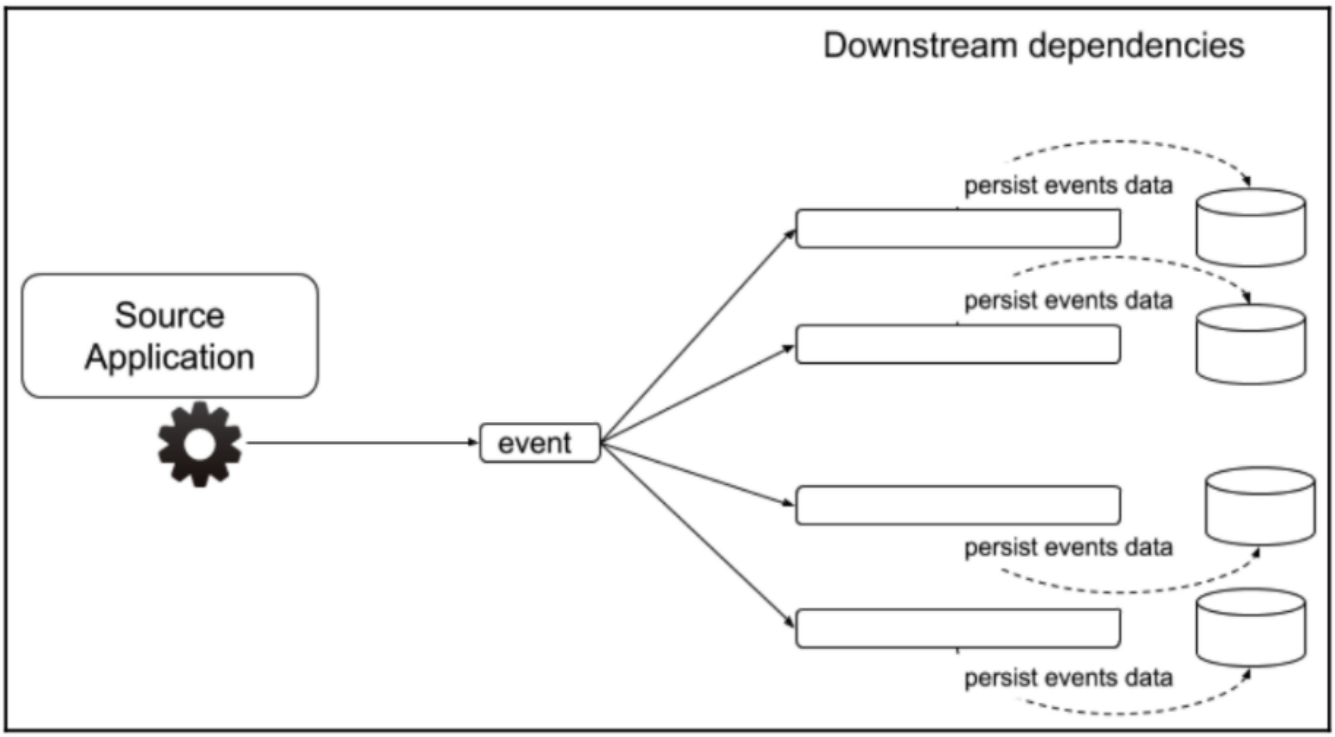



Decoupled systems and inverted dependencies sound fantastic, but the implicit disadvantage of these is that you lose visibility. This is because the application emitting events doesn't know anything about the processes that are executed once the event is published, and there is no code for reading other systems.

It is often impossible to identify dependencies downstream, and some techniques for correlating events across different logs are commonly used to relieve this nightmare.

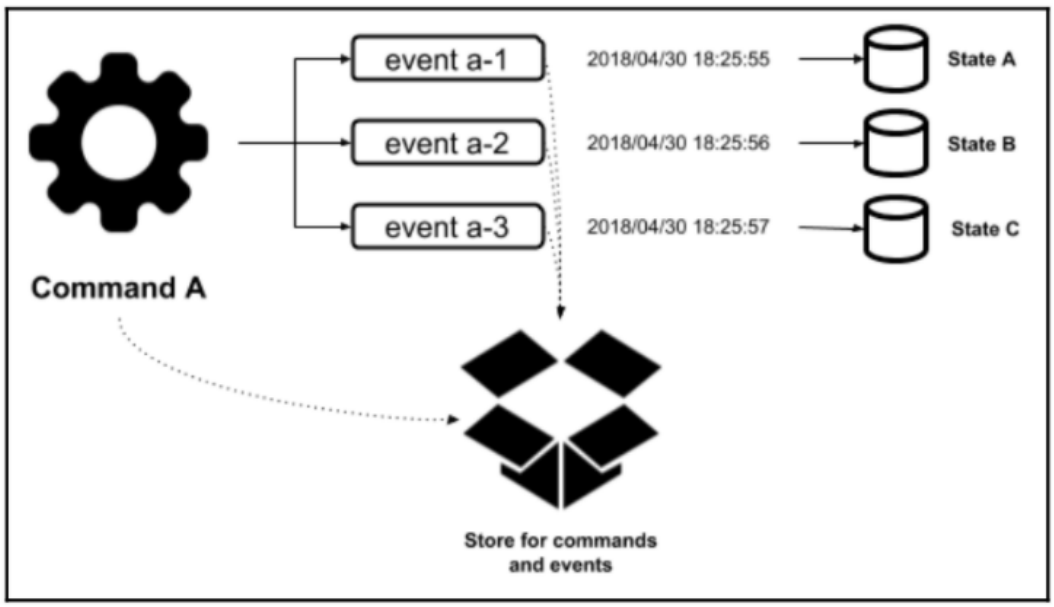
Coupled systems give all the information regarding downstream dependencies, and are hard to evolve. Conversely, decoupled systems know nothing about downstream dependencies, but they offer the chance to evolve systems independently.

## Event-carried state transfer




**Event souring:**

Let's review the following set of diagrams to understand how this works. The first diagram shows that once `Command A` is executed, three `Events` are created, and a new `State` is generated after each one of them is processed:




We mentioned that an event-sourcing system should have *at least* two places in which to store the data. One of these will be used to save event and command information and the other one will be used to save the application state—we say *at least two* because more than one storage option is sometimes needed to persist the system state of an application. Since the input retrieved by the system to perform their business processes are very different from each other, we should consider using a database that supports the ability to store data using a JSON format. Following this approach, the most basic data that should be stored as part of a command that is executed within an event-sourcing system is as follows:

- Unique identifier
- Timestamp
- Input data retrieved in JSON format
- Any additional data to correlate commands

Event sourcing:

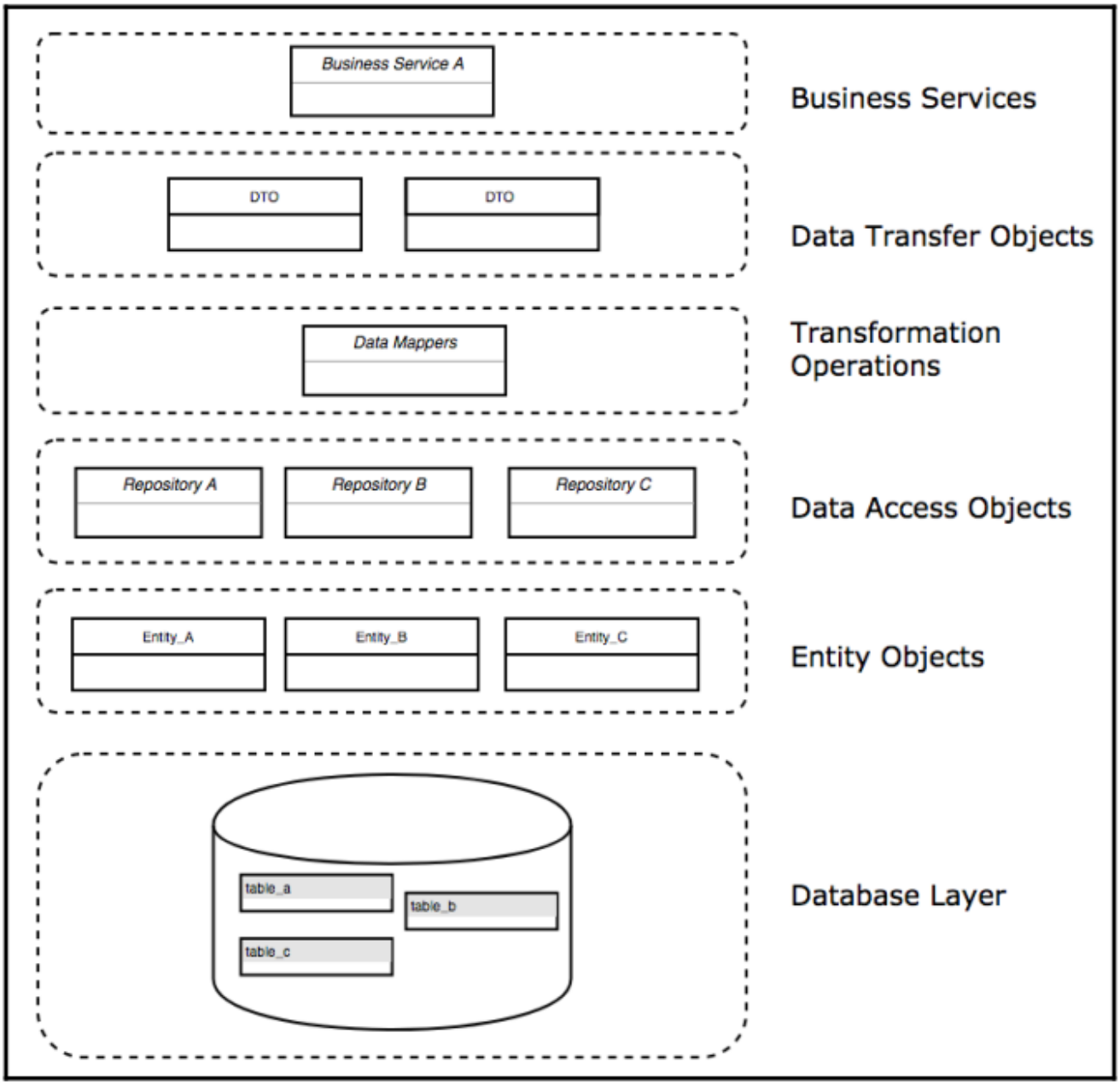
- Require at least 2 data store, one for saving events, one for saving application state (traditional database)
- We can re-construct application state from events

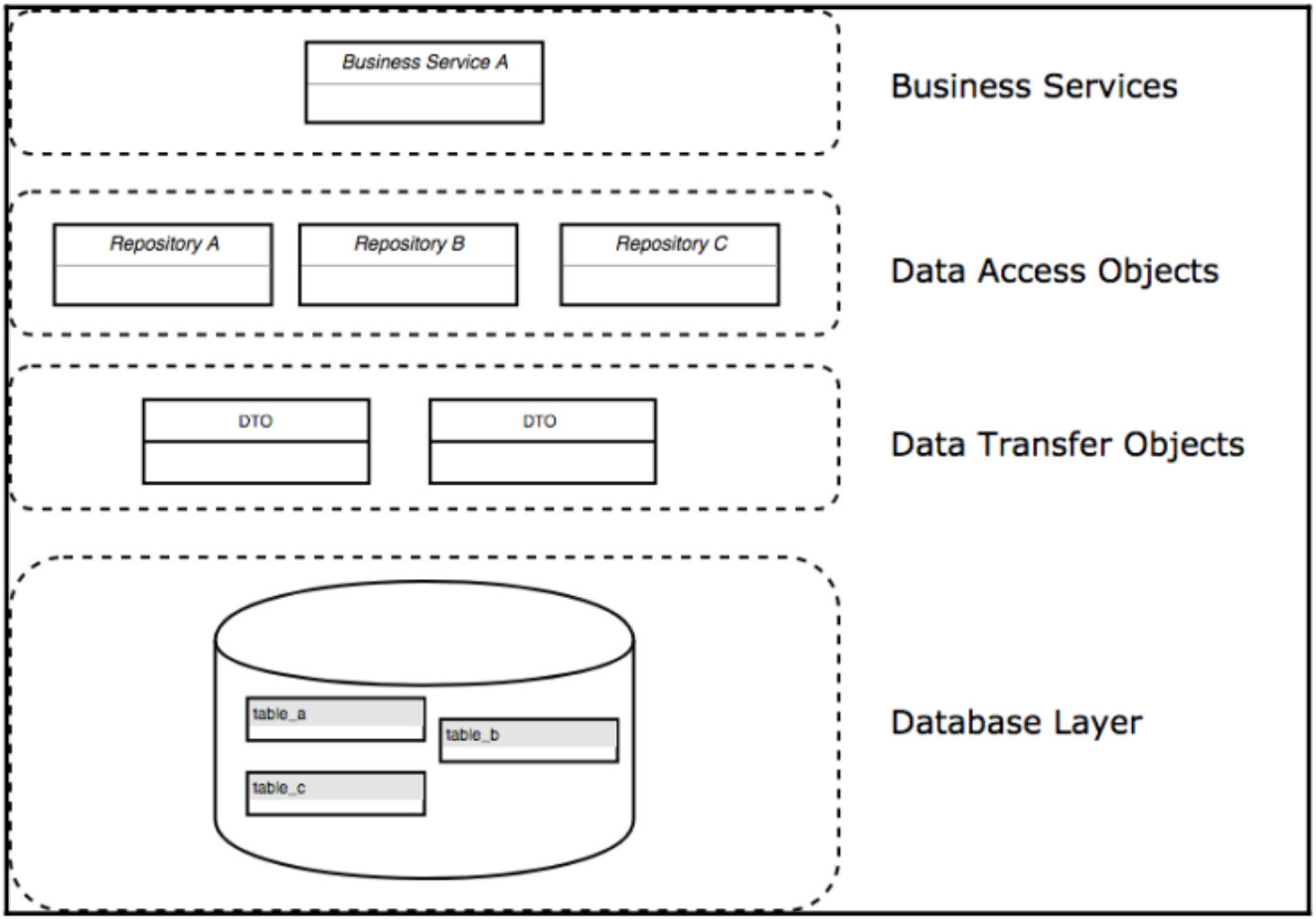
CQRS

Why need CQRS:

- Complex domain model

- Seperate scaling
- Query vs command seperation



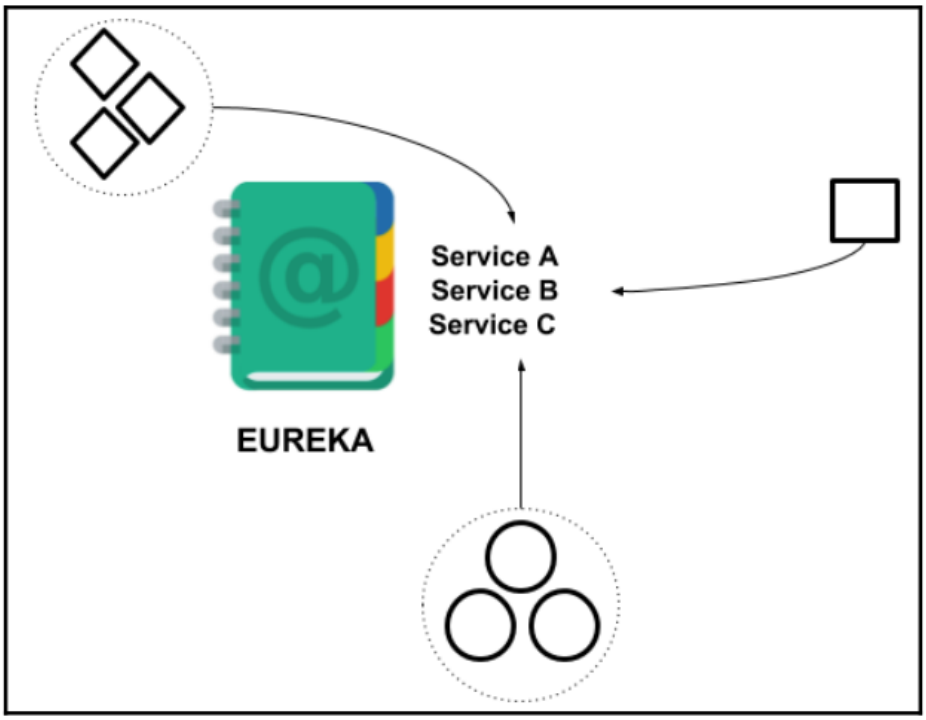



Data flow using DTOs

Transforming data is not a big deal. In systems using an ORM, the biggest problem arises when entity objects bring columns containing useless information that is ignored during the transformation process, introducing an unnecessary overhead on the database and the network. On the other hand, in the preceding diagram, we can see that a big process is needed to map the database tables as objects before actually getting the requested data. A good approach to getting rid of this problem is to replace the read operations executed by the ORM frameworks with stored procedures or plain query statements to retrieve only the required data from the database.

## Microservices:

### Eureka:

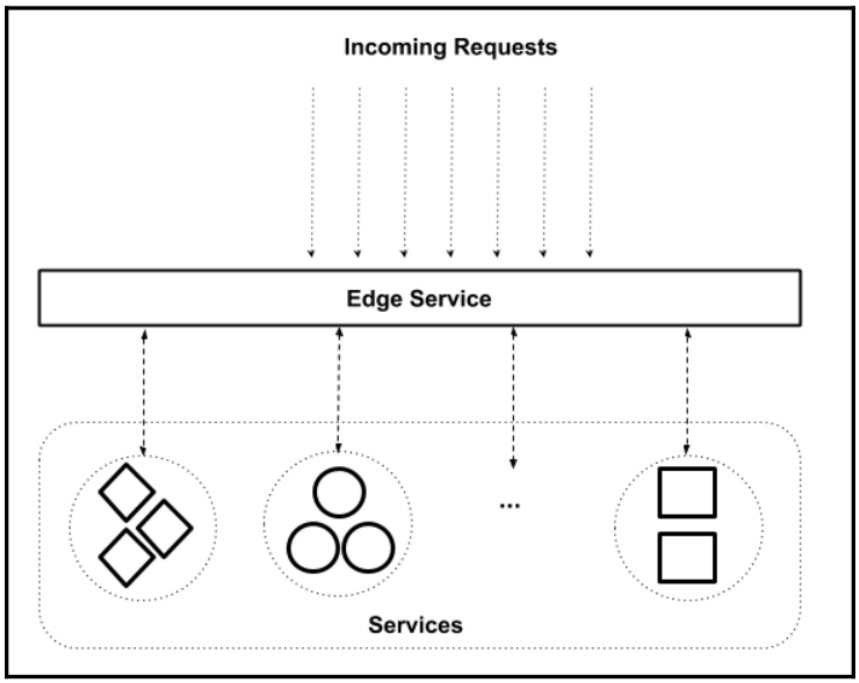



- Build eureka server
- Client registry to server

**Ribbon:**

- Client side load balancing and discovery

**Edge Services:**

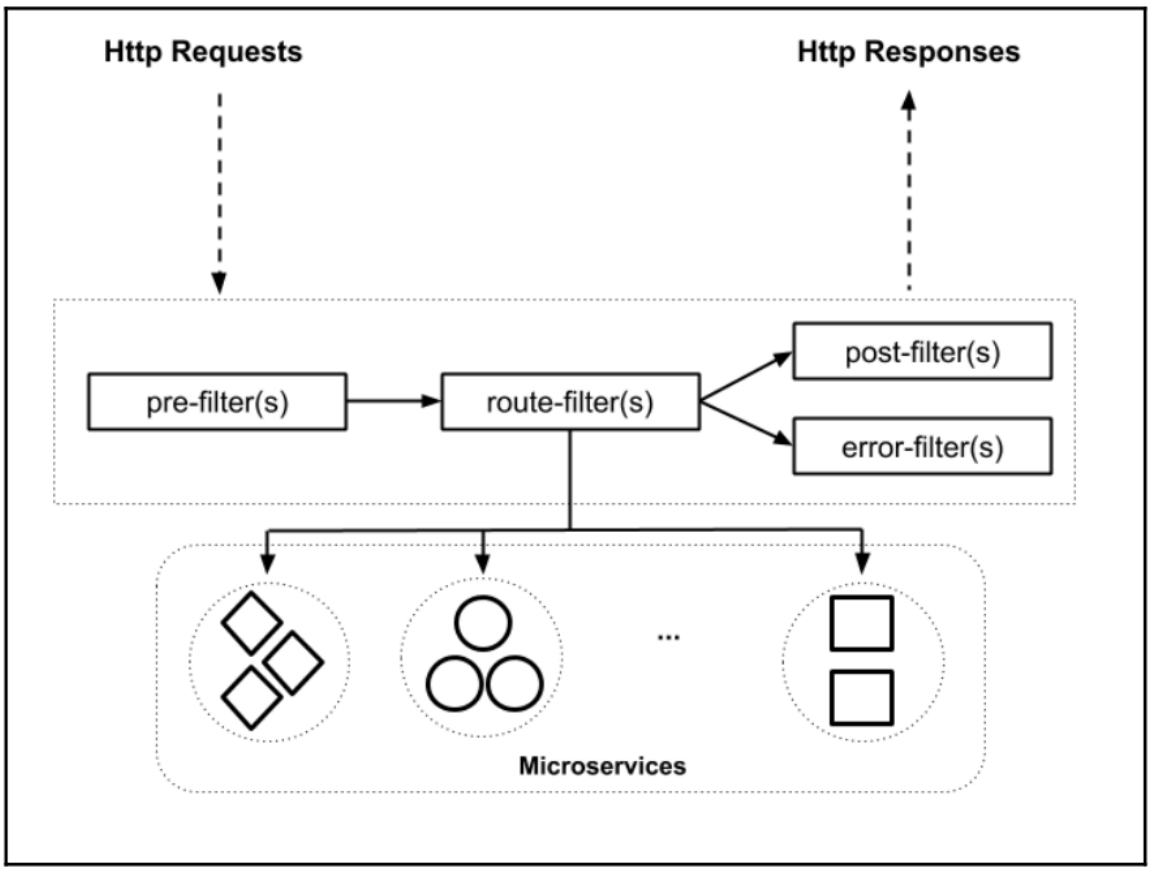



Example:

- Resolve CORS
- Authentication

**Edege Zuul Services:**

You can apply the filter during four phases, as shown in the following diagram:

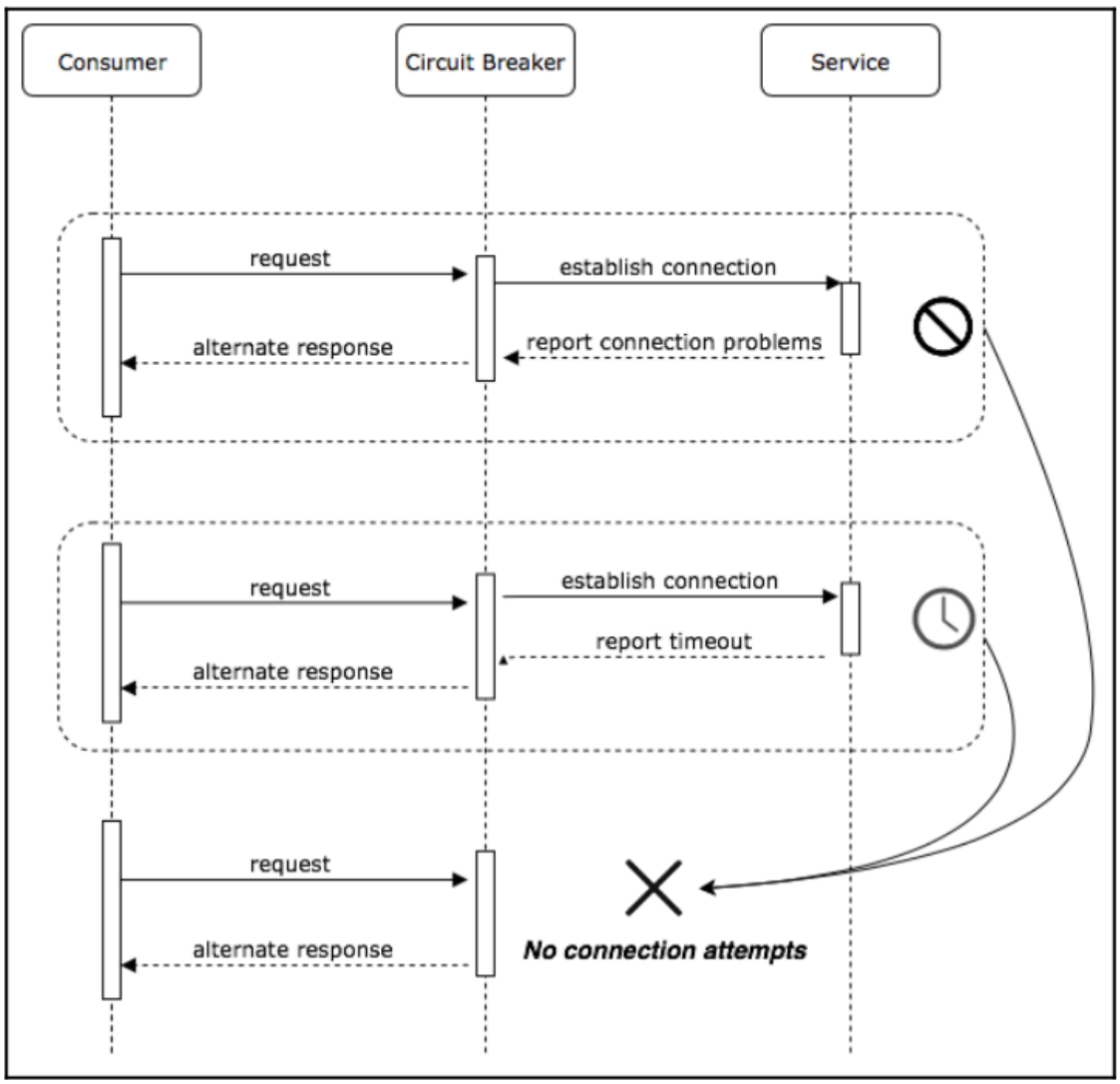



Edge service used case:

- Authentication
- Authorization
- Rate limits
- Translation and transformation operations in the request body
- Custom headers injection
- Adapters

**Circuit breaker**

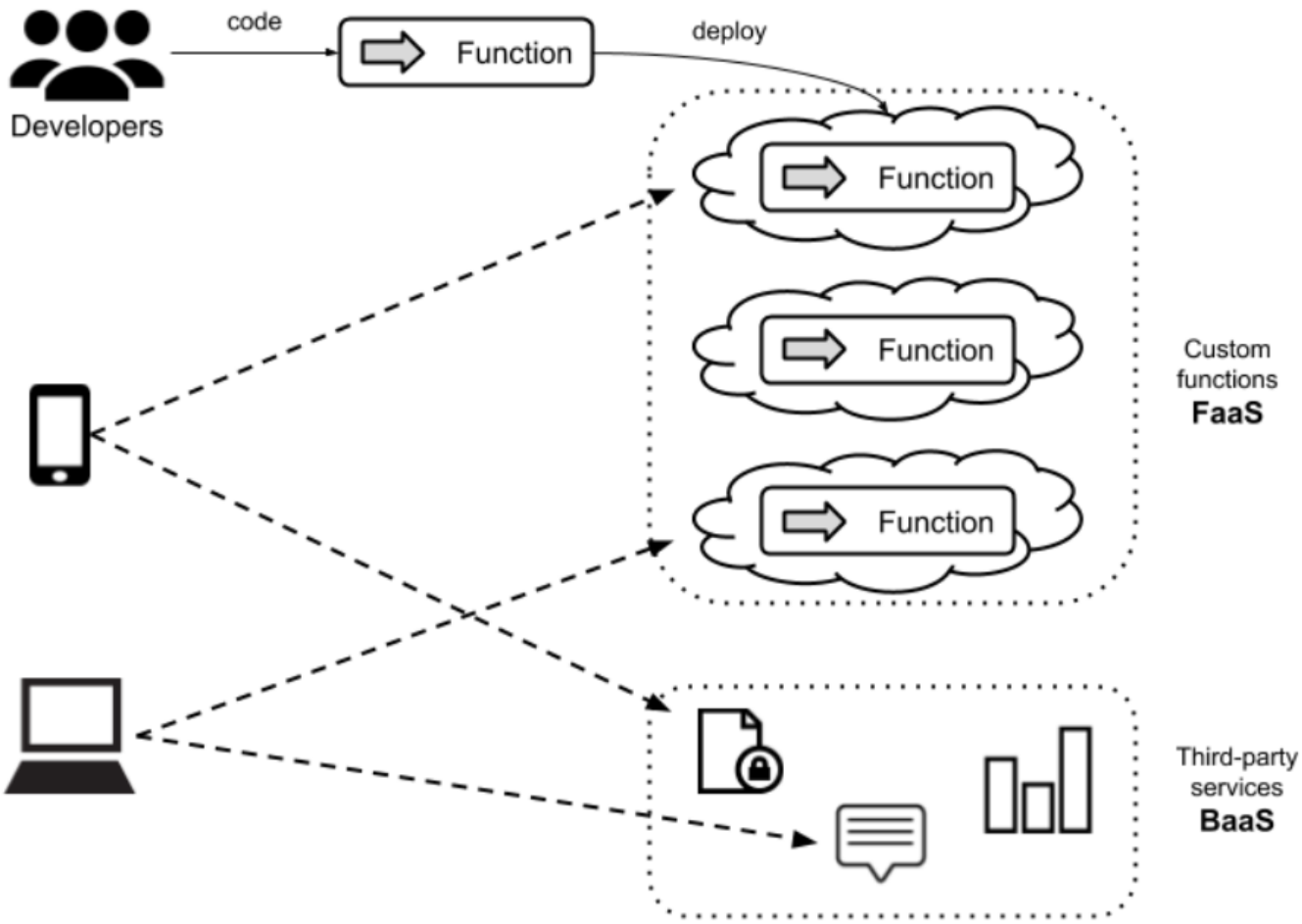
The circuit breaker pattern is intended to handle failures that are created when a system interacts with other systems that are running in different processes using remote calls. The main idea behind this pattern is to wrap the call with an object that is able to monitor failures and produce successful responses, as shown in the following diagram:




## Serverless Architectures

Common examples of serverless and FaaS include:

- Authentication
- SMS notifications
- Email services




Some common scenarios that serverless architectures can be applied to are as follows:

- Processing webhooks
- Tasks or jobs that should be scheduled or triggered under certain circumstances
- Data transformation, for example:
    - Image manipulation, compression, or conversion
    - Voice data transcribed into text, such as Alexa or Cortana
- A certain logic for mobile applications, based on the mobile backend as a service approach
- Single-page applications
- Chatbots

On the other hand, serverless architectures are not suitable for the following scenarios:

- Long-running processes where huge amounts of resources (such as CPU and memory) are required
- Any blocking processes

## Monitoring:

- We can use an old-fashioned approach that implies creating interceptors around methods to log everything we want around them.
- Spring Actuator can be used along side Spring Boot applications. Using this library, we can review the health of an application; it provides an easy way to monitor applications via HTTP requests or JMX. Additionally, we can use tools to index the data produced and to create graphs that are helpful to understand the metrics. There are plenty of options to create graphs, including:
    - ELK Stack (ElasticSearch, Logstash, and Kibana)
    - Spring-boot-admin
    - Prometheus

**Perfomance:**

- Visual VM: use for monitor CPU, threads
- SQL query optimization: review SQL, use batch processing, conncetion pool, avoid ORM join.
- Load test