**Chapter 3. Connectivity and Composition Patterns**

Cloud native applications are inherently a distributed collection of microservices connected via interservice communication. When building real-world cloud native applications, it is critical to establish interservice connectivity, to integrate multiple services to create business capabilities, and to present these connected services as managed capabilities.

In this chapter, we explore a wide range of patterns that we can use to build connectivity among microservices (as well as with other existing systems) in a cloud native application. We also look at creating business functionalities by using *service composition* patterns to integrate services. Let’s begin our discussion with connectivity patterns for building cloud native applications.

**Connectivity Patterns**

native *connectivity patterns* allow you to build connectivity among microservices as well as with the other systems in your cloud native application. As we discussed in [Chapter 1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#introduction_to_cloud_native), a cloud native application consists of microservices and may also connect with existing proprietary or legacy systems, external services such as software-as-a-service (SaaS) applications, databases, messaging infrastructure such as message brokers, and more.

**NOTE**

In [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns), we primarily discussed how interservice communication takes place between microservices of a cloud native application. In this section, our main focus is on the styles that we can use to connect those services. Here we will use foundational communication patterns as well as other supporting technologies to connect our microservices when building cloud native applications.

In addition to the connectivity requirements related to the business capabilities of your cloud native application, you may need to consider the connectivity requirements for nonfunctional capabilities such as security, observability, reliability, and load balancing. This requires you to connect the microservices of your cloud native application with other systems such as identity management services and observability tools. Therefore, having the ability to seamlessly and efficiently connect all these services and systems is important when building cloud native applications. In this section, we’ll explore key patterns that are useful when building the connectivity between your microservices and other systems.

**Service Connectivity Pattern**

The *Service Connectivity pattern* is a high-level, composite pattern that can be used in building cloud native applications. This pattern explains how a cloud native application is formed by connecting microservices and existing systems, and how these services interface with the consumers of the application.

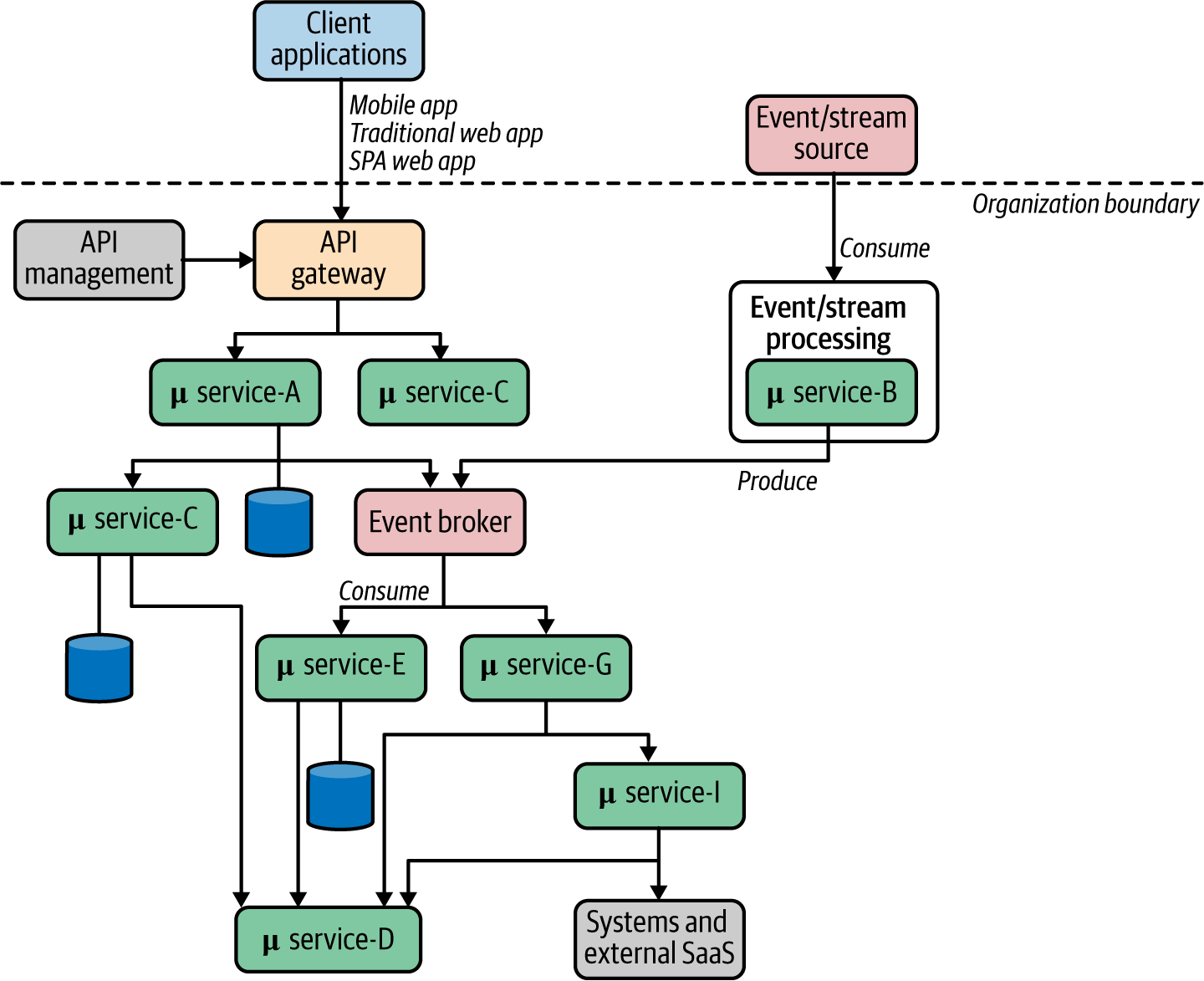
**How it works**

The Service Connectivity pattern provides a generic way to connect different components such as microservices, external systems, and APIs exposed to consumers. This pattern uses one or more foundational communication patterns such as synchronous or asynchronous communication to establish the connectivity in your cloud native applications. It is applied to the backend implementation of any of the business capabilities of your cloud native application. We explore the patterns related to frontend and backend connectivity later in this chapter.

You can mix and match the most suitable communication patterns to build the Service Connectivity pattern ([Figure 3-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_connectivity_pattern-id00204)). These services can connect with other systems such as databases, message brokers, or any external system. The business capabilities that we choose to present to consumers are exposed as managed APIs using the API gateway layer.

In [Figure 3-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_connectivity_pattern-id00204), microservices are connected to multiple other services via synchronous communication. These services can use patterns such as Request-Response or RPC to build the connectivity. Other services may require asynchronous communication via an event or message broker to build queue-based Single-Receiver communication or publisher-subscriber-based Multiple-Receiver communication.

In addition to these interactions, the microservices need to connect with external services and databases. The API management layer that sits on top of all the business capabilities of a cloud native application makes sure that all these capabilities are offered as managed APIs to consumers. (We’ll dive into API management patterns in [Chapter 7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_and_consumption_patterns).) In certain cases, you may also want to apply techniques such as event stream processing to process data from external and internal event-streaming sources. ([Chapter 6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#stream_processing_patterns-id00204) covers these patterns in detail.)



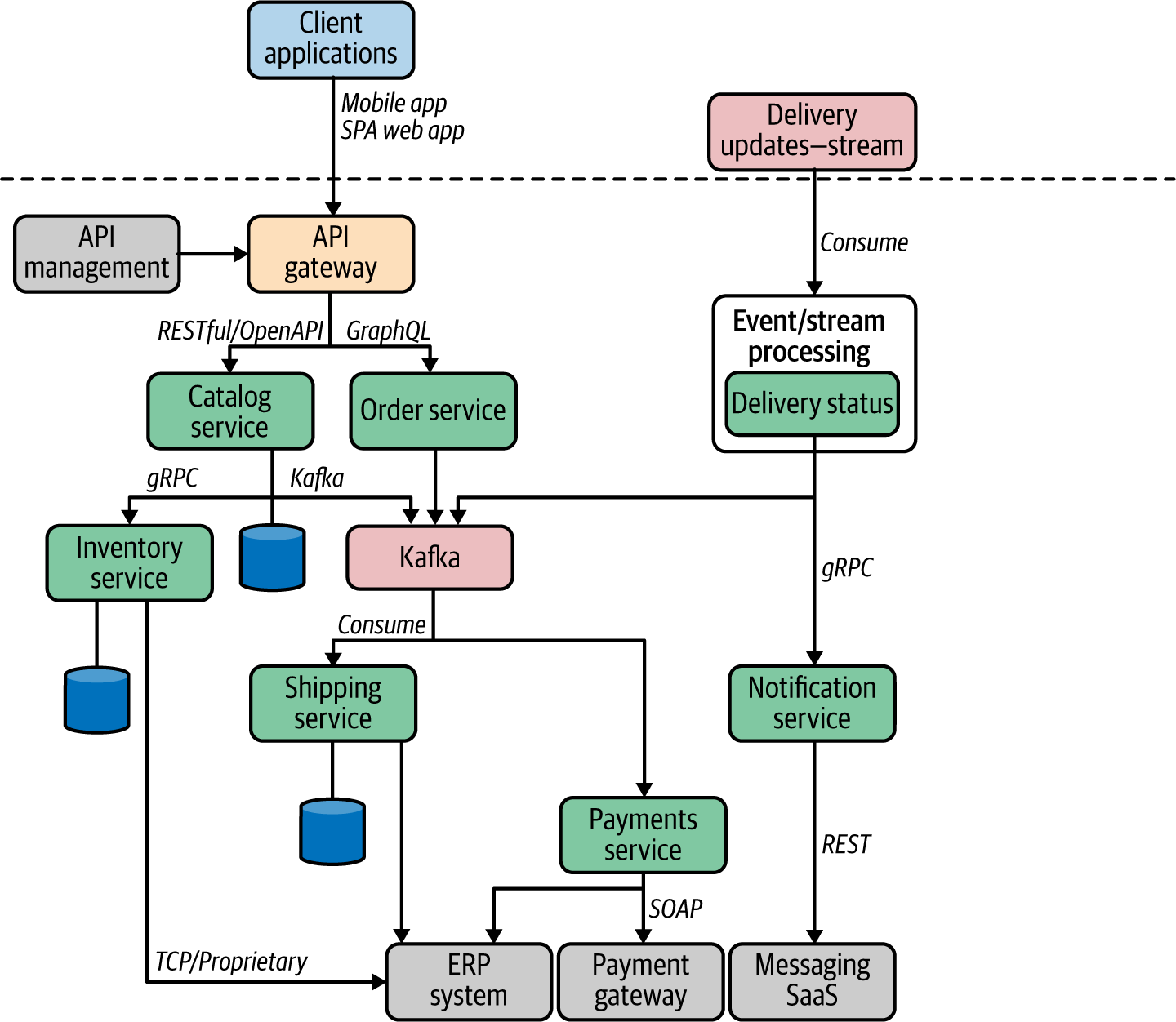
**Figure 3-1. Service Connectivity pattern**

So, the key idea of the Service Connectivity pattern is that it should be used as a high-level pattern that connects services and systems to realize the end-to-end business capabilities of a cloud native application.

**How it’s used in practice**

As we discussed earlier, the Service Connectivity pattern is used in almost all cloud native applications. Any application that has more than one microservice or system needs to use this pattern to connect them and build the business capabilities. In most real-world use cases, the Service Connectivity pattern leverages several foundational communication patterns to connect the required microservices and other systems.

[Figure 3-2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#using_the_service_connectivity_pattern) shows a sample use case of an online retail application built using this pattern. As the Service Connectivity pattern outlines how to connect services and systems using foundational communication patterns, this example uses multiple communication patterns to build various parts of the use case. For instance, communication with external-facing services (Catalog and Order) inherently use request-response messaging using REST and GraphQL, while some of the internal service interactions (communication between the Order and Payments services) were done asynchronously using a Kafka broker.



**Figure 3-2. Using the Service Connectivity pattern in an online retail application**

An API gateway is used to expose our online retail application’s APIs to external consumers such as mobile clients or web portals. Most of the internal microservice connectivity is implemented with RPC using protocols such as gRPC. The required proprietary and legacy systems are used to build the business capability by calling them through microservices. For example, the Inventory service calls a proprietary enterprise resource planning (ERP) system in this example. Certain parts of the application, such as the Delivery Status service in this scenario, consume an external event stream and process those events.

**Considerations**

As you can see in [Figure 3-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_connectivity_pattern-id00204), the more microservices and systems that we have to connect, the greater the complexity of the entire cloud native application. Therefore, before applying this pattern, we need to make sure that we are using the appropriate service granularities. If you are seeing too many interactions among microservices, that’s a sign that your service design scope is too fine-grained. If that’s the case, you need to revisit the microservice design phase again and redefine the scope of the services so that they are directly mapped to the business capabilities, rather than framed features or utilities.

Also, as we discussed in [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns), the foundational communication patterns used in our application need to be determined by the business use case. For example, an interactive business capability such as searching for items in an online retail store needs a synchronous Request-Response communication pattern, while placing orders requires a more durable guaranteed delivery messaging pattern such as the asynchronous Single-Receiver pattern. Therefore, we need to spend quality time identifying or defining service interaction styles and the underlying protocols to use. This is usually something that you will do as part of the microservice design phase itself. When it comes to third-party systems, you don’t have much control over them and will have to rely on the interaction interfaces and protocols they offer.

When building service interactions, we need to make sure that we don’t leak any infrastructure (or anything else that is not related to business logic) to our application’s connectivity logic. If we do, our application’s business logic gets coupled to the infrastructure or the environment, and we lose the application’s portability. The service connectivity capabilities that are not mandatory for the application behavior need to be implemented at other layers (such as sidecars), which we will explore later in this chapter.

**Related patterns**

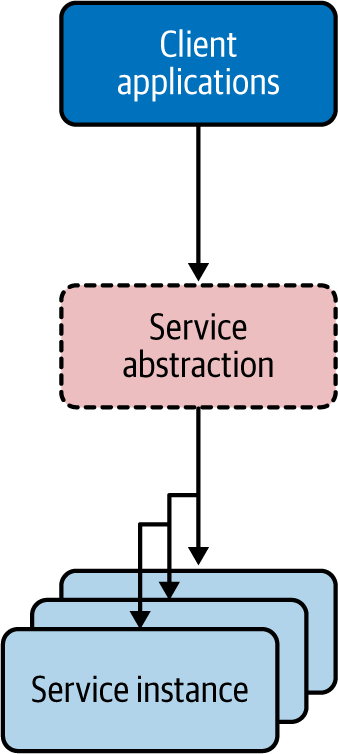
The Service Connectivity pattern provides a high-level view of how a typical cloud native application is formed by connecting its microservices and supporting systems. Most of the foundational cloud native communication patterns covered in [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns) as well as other patterns that we’ll explore throughout this chapter are closely related; we use them inside the Service Connectivity pattern.

**Service Abstraction Pattern**

When a microservice in a cloud native application needs to interact with another microservice or an external system, it is preferable to use an abstraction that hides the details of the underlying implementation, location, and deployment structure. That is the key idea behind the *Service Abstraction pattern*. This pattern uses a service to abstract one or more underlying services.

**How it works**

A given microservice or any other external system in a cloud native application can be represented as a service, so that it hides all the implementation details. For example, a given service may have multiple runtime instances running in different locations, with different Domain Name System (DNS) names and Internet Protocol (IP) addresses, and so on. If we don’t use an abstraction to represent that microservice, all the clients or other services that consume it need to know the implementation details of the target service or system. Therefore, we can introduce a service abstraction in front of the microservices or systems in a cloud native application ([Figure 3-3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#the_service_is_an_abstraction_that_hide)).



**Figure 3-3. The service is an abstraction that hides the implementation details of the underlying microservice or the system**

A service groups a set of runtime instances of the same microservice or the system (it can even be a monolithic system that our cloud native application needs to interact with). The underlying instances can come and go, and their IP addresses can change without affecting consumers. Here are some of the key benefits of using the Service Abstraction pattern:

* It allows you to use a stable or fixed location (IP) to represent your microservice or system within the cloud native application.
* It can provide a built-in service-discovery capability, so that the consumer applications refer to the service by using a generic naming scheme that hides the implementation details (for example, *http://hostname:port/checkout*). This is the key idea behind the Service Registry and Discovery pattern, which allows us to represent all the microservices and systems in our cloud native application at a central location so other consumers can discover existing services or register new services.
* It can seamlessly provide load balancing and failover.
* Dynamic scaling of the microservice or system is possible when using a service abstraction, as the underlying instances can come and go as needed.

Let’s look at how the Service Abstraction pattern is used in cloud native applications.

**How it’s used in practice**

The notion of the service abstraction is often built along with the service or the system. Using a service abstraction for any service instance or system has been used even in the SOA era. However, it was widely adopted with the rise of containers (Docker) and Kubernetes.

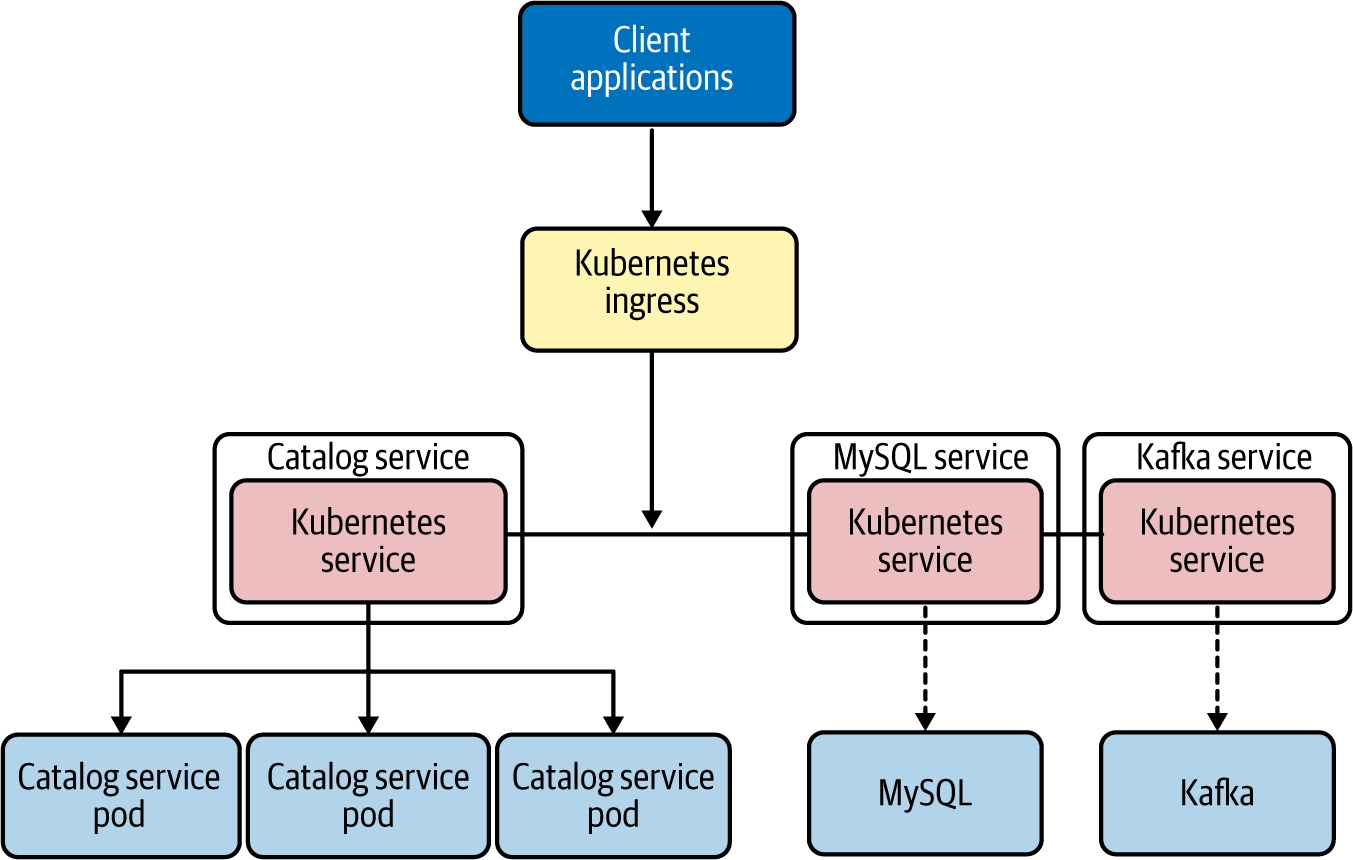
**Kubernetes services**

Certain platforms, such as Kubernetes, use a service abstraction as a fundamental construct. This makes life easier for the developers to represent all the microservices and systems that need to interact with Kubernetes services.

A Kubernetes service groups a set of pod endpoints into a single resource. You can configure how you access that service in various ways (for example, load balancing or cluster IP). [Figure 3-4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#kubernetes_group_a_set_of_podscomma_or) shows how the Catalog service abstraction deployed on Kubernetes groups Catalog service instances into pods. It exposes a stable endpoint to consumers, while we can dynamically change the location and number of pods backing the Catalog service. With the Kubernetes Catalog service of the online retail application, you get a stable cluster IP address that clients inside the cluster can use to invoke the service. A client sends a request to the stable IP address in the Kubernetes cluster, and the request is routed to one of the pods in the Catalog service.

The Kubernetes service also provides seamless load balancing among the grouped pods. Kubernetes allows you to define various types of Kubernetes services to control the way you expose your microservice or any other system. For instance, if you use the service type LoadBalancer, Kubernetes automatically creates a cloud network load balancer of the underlying cloud platform (AWS, Azure, GCP, and so forth). This load balancer provides an externally accessible IP address that sends traffic to the correct port on your cluster nodes, provided your cluster runs in a supported environment and is configured with the correct cloud load-balancer package.

As we discussed earlier, with the Service Abstraction pattern, you can expose any other monolithic or proprietary systems that are consumed by your cloud native application as a service as well. For example, Kubernetes allows you to use the service type ExternalName that provides an internal alias for an external DNS name. As shown in [Figure 3-4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#kubernetes_group_a_set_of_podscomma_or), if there’s a monolithic deployment of a Kafka broker, you can represent it as an external system and consume that as a Kubernetes service for your microservices.



**Figure 3-4. Kubernetes groups a set of pods, or any external system, to a single abstraction known as a service**

**Considerations**

Service Abstraction is more or less a mandatory pattern that you need to use when building cloud native applications. The power of the service abstraction is critical to achieve the scalability, redundancy, and encapsulation (hiding the implementation details) of your microservices and other systems. While implementation of the Service Abstraction pattern without using an underlying platform is possible, we recommend using a platform such as Kubernetes that already supports service abstractions as a first-class construct.

**Related patterns**

Service Abstraction is commonly used with most of the connectivity patterns in this chapter. For example, service abstractions are used in the Service Registry and Discovery pattern, which we discuss next.

**Service Registry and Discovery Pattern**

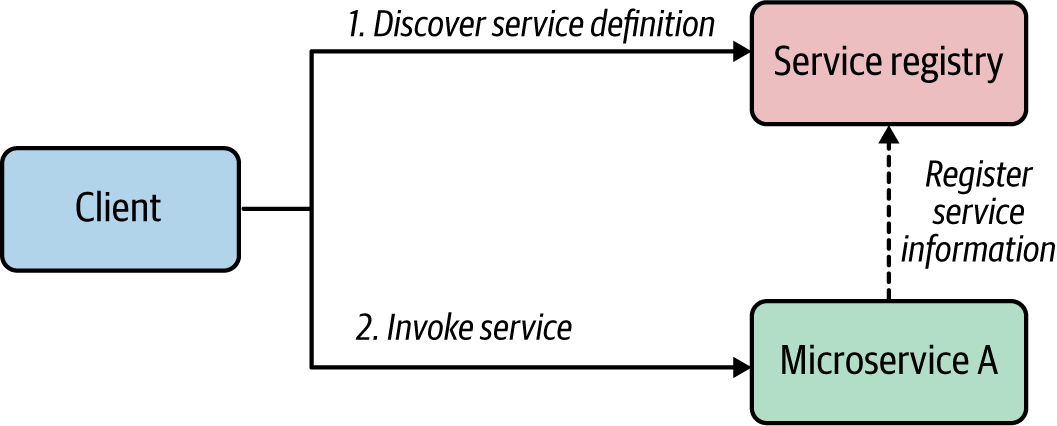
When you build cloud native applications, you need a place to keep the information about the services that you create. This enables consumers to find out all the details of the services. The *Service Registry and Discovery pattern* can be used for this.

**How it works**

Once you represent all the entities in a cloud native application as services (through the Service Abstraction pattern), you need to keep information about these services so consumers can obtain that data and access them. The repository containing this service information and metadata is the service registry. Often we keep service information such as service URLs, service interface definitions (for example, OpenAPI specs or gRPC Protobuf definitions), service-level agreements, and other information that is useful to the service consumers. A service registry typically uses a canonical representation of a service so that we can define any service metadata at the service-registry level, despite the technologies that we use to implement them.

A service registry is implemented as another service that offers a registry repository API and discovery API. The service consumers/client applications can get the service information by accessing the service registry API ([Figure 3-5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_registry_and_discoveryem_dashcl)). Service owners/developers can register the service by providing its details. The owners are responsible for updating and maintaining the service information at the service registry. The service consumers then use that service registry to learn how to consume the service.

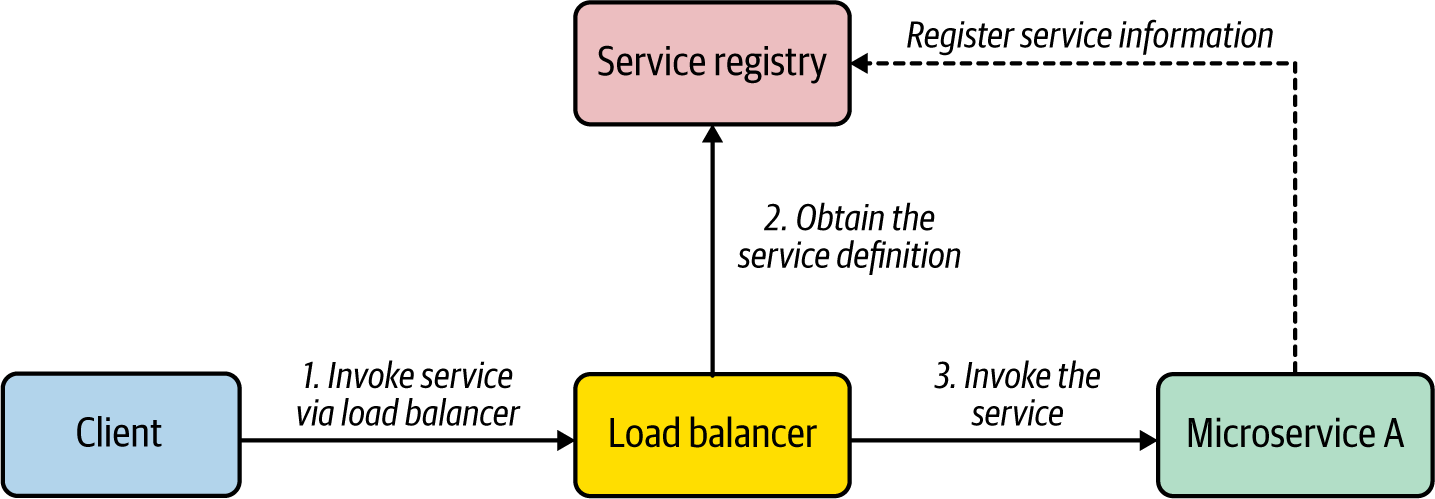
We can implement the Service Registry and Discovery pattern in two ways. In the first approach, called *client-side service discovery*, the client is responsible for service discovery ([Figure 3-5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_registry_and_discoveryem_dashcl))*.*



**Figure 3-5. Service Registry and Discovery—client-side discovery**

The information that we store in the service registry can be a service definition or service contracts. Once consumers obtain this information, they can invoke the corresponding service by processing the service information. The consumption of the service registry can happen during both runtime and development time. We can use it at runtime to determine the endpoint address or security policies of a given service, and we can use it at development time to obtain the service contract and build the consumer application according to the service contract.

In certain scenarios, we can offload the service discovery task to an intermediate component such as a load balancer. This service discovery mechanism is known as *server-side discovery.* In this scenario, the consumer/client simply sends the request to the load balancer with a reference to the service that it wants to invoke and the corresponding message ([Figure 3-6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_registry_and_discoveryem_dashse)).

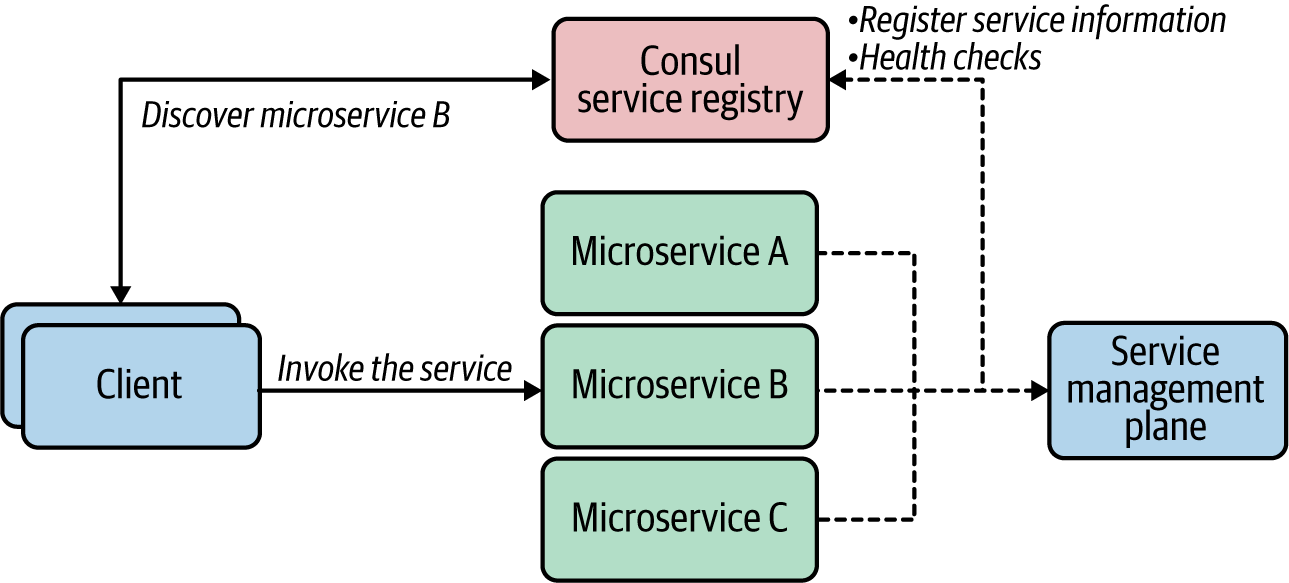


**Figure 3-6. Service Registry and Discovery—server-side discovery**

The load balancer is preconfigured to work with the service registry so that when a request is made for a given service, the load balancer retrieves the service information from the registry (such as the service endpoint URL) and uses it to invoke the service. (In most cases, the load balancer caches the service information rather than invoking the registry per each request.)

**How it’s used in practice**

Service Registry and Discovery is a mandatory pattern for building any real-world cloud native application because of the immutable nature of microservices. To use this pattern in practice, we can use a dedicated service registry implementation such as Consul, as shown in [Figure 3-7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_registry_and_discovery_with_con). All the microservices need to register with the service registry at the time of deployment. We can also configure them to send heartbeats to the registry to detect any unavailable services.



**Figure 3-7. Service Registry and Discovery with Consul**

If you use a separate service management or governance plane to manage the service life cycles, you can use the service registry as the service repository for that management plane. The service registry can be configured to work with either client-side discovery or server-side discovery using a load balancer.

**Service discovery in Kubernetes**

If you are using Kubernetes to invoke one service from another service, you don’t need to worry about the actual location of the service that you are invoking. Kubernetes by default uses DNS names to discover the pods. Therefore, if you want to call the Bar service from the Foo service, in the Foo service’s code you can just refer to *http://bar:<port>* as the service endpoint. Kubernetes will resolve and map the name to the actual endpoint. Kubernetes internally uses etcd as the distributed key-value store that is used as the service registry. However, this built-in registry component does not provide the same level of capabilities for managing the service metadata as a dedicated service registry such as Consul. Therefore, if you have complex service registry and discovery requirements, you can use a dedicated service registry alongside Kubernetes.

**Considerations**

Service Registry and Discovery is an essential pattern for building cloud native applications. However, this doesn’t mean you should have a full-blown service registry and discovery solution from day one. For the most part, the primitive capabilities of Service Registry and Discovery are offered in the platforms used to build cloud native applications, such as Kubernetes. If you are using a cloud service, such as AWS, Azure, or GCP, Service Registry and Discovery will be supported as part of the cloud service. Therefore, you should invest in a dedicated service discovery and registry solution if the your use case absolutely requires you to have advanced Service Registry and Discovery capabilities such as managing service dependencies and associations, health checks, and leader election.

**Related patterns**

Service Registry and Discovery is a foundational pattern used along with most of the connectivity patterns in this chapter. In the context of API management, when you have to expose certain capabilities as APIs, we use a dedicated API registry known as the *API developer portal*, which is similar to a service registry. But it’s a dedicated API repository to store the business capabilities that you expose to your consumers, and not all services are published to the developer portal.

**Resilient Connectivity Pattern**

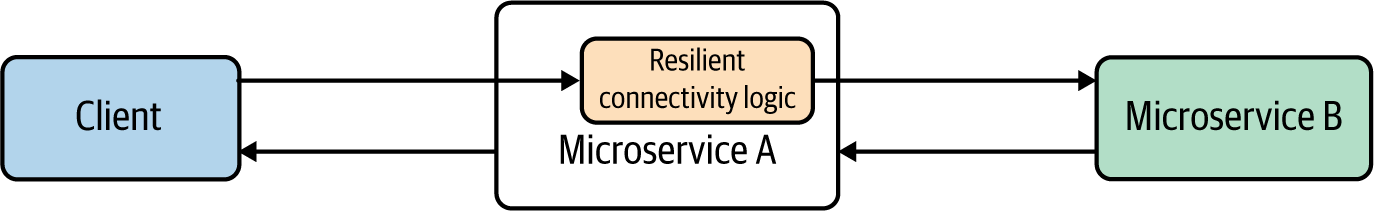
When you are building connectivity among microservices as well as the other systems in your cloud native application, you need to use a network. In distributed computing, the network is always considered to be unreliable. Therefore, we need to make sure that we connect microservices and systems by using resilient connectivity techniques.

**How it works**

The *Resilient Connectivity pattern* allows you to design a resilient interaction between the microservice and the other services or systems it invokes, so that if a failure occurs, the system will be able to handle it or recover from it. For example, suppose you have two microservices in your cloud native application: Microservices A and B. Microservice A invokes Microservice B via network communication, and we need to make sure that communication happens resiliently.

The logic that invokes Microservice B should be able to handle the failures that could occur, recover from them if possible, or gracefully take actions to avoid the failure in the future. The key idea here is that Microservice A contains resilient communication logic that is executed as part of the service runtime.

Depending on the nature of the failure that could occur during interservice communication, we may implement the resilient communication logic in different ways, but the high-level architecture implementing resilient communication can be generalized as shown in [Figure 3-8](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#calling_microservices_and_systems_resil).



**Figure 3-8. Calling microservices and systems resiliently**

**How it’s used in practice**

The Resilient Connectivity pattern is implemented to handle multiple communication requirements. Let’s discuss the various styles of resilient communication.

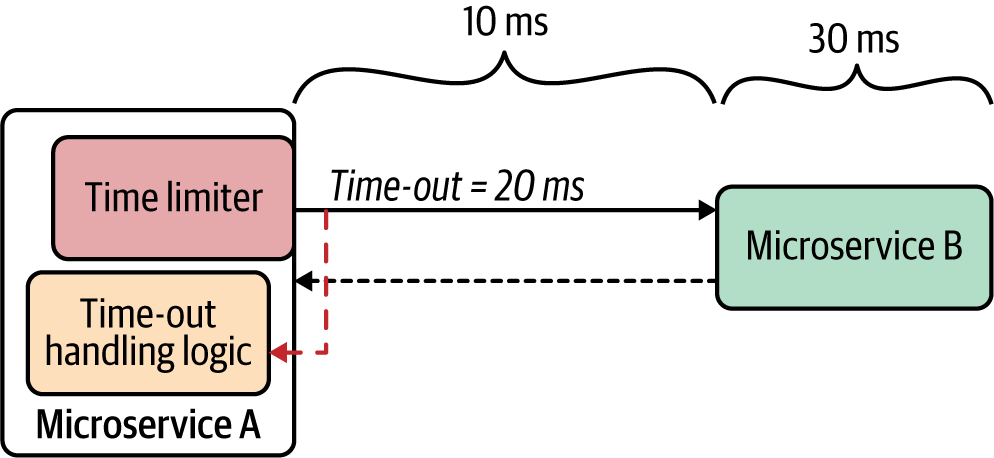
**NOTE**

The various applications of the Resilient Connectivity are sometimes defined as separate patterns. For more information, consider reviewing the patterns defined by Michael Nygard in [*Release It!*](https://www.oreilly.com/library/view/release-it/9781680500264) (Pragmatic Bookshelf).

**Time-out**

A *time-out* is used when one service calls another one and waits for a timely response or acknowledgment. If we don’t use a time-out when invoking another service or system, the caller service waits indefinitely for a response from the target service. That behavior hinders the responsiveness of the cloud native application; even if a failure occurs, the application takes an indefinite time to detect it. So, if we implement the caller service’s connectivity logic that can decide when to stop waiting for a response, we call that time duration the *time-out*. Once we reach a time-out on the caller side, we can specify time-out handling logic that can gracefully handle the situation.

In [Figure 3-9](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#resilient_connectivity_with_time_outs), Microservice A calls Microservice B by using resilient communication logic that has a time limiter specifying a time-out of 20 milliseconds (ms). However, the network latency is 10 ms, and Microservice B’s processing time is 30 ms. Since the round-trip time of 40 ms (30 + 10) is greater than the time-out (20 ms), the time-out handling logic is invoked.



**Figure 3-9. Resilient connectivity with time-outs**

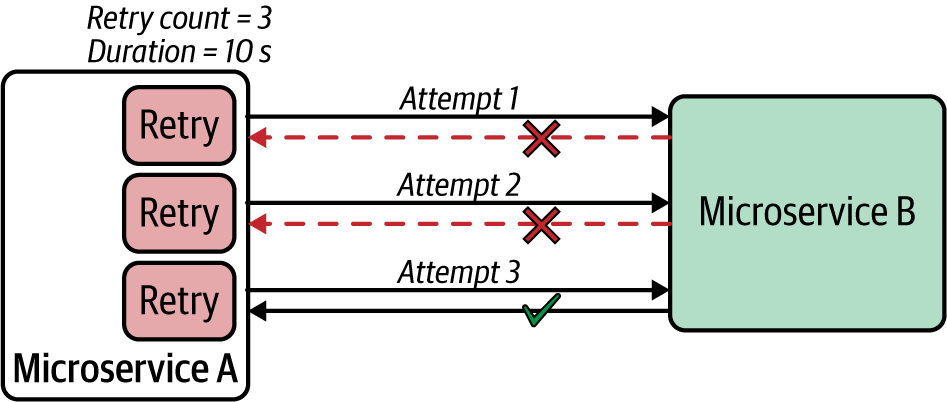
We need to be realistic when setting time-out values, because if the time-out is too low or too high, we won’t get satisfactory results. Consider the typical network latency and target service’s processing time prior to setting the time-out value.

A time-out helps services isolate misbehavior or the anomaly of another service or system, so it does not have to become your service’s problem.

**Retry**

When services communicate over a network, intermittent failures can occur. The key idea behind retry logic is to provide a way to get the expected response, despite network disruption, after trying to invoke the same service one or more times. As part of the *retry* resilient connectivity logic, we can specify the number of total retries that the service should invoke and the duration between retries.

[Figure 3-10](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#retry_logic_trying_to_invoke_the_same_s) shows Microservice A calling Microservice B with retry resilient connectivity logic. Microservice A will trigger three retries in 10-second intervals until it successfully receives the response.



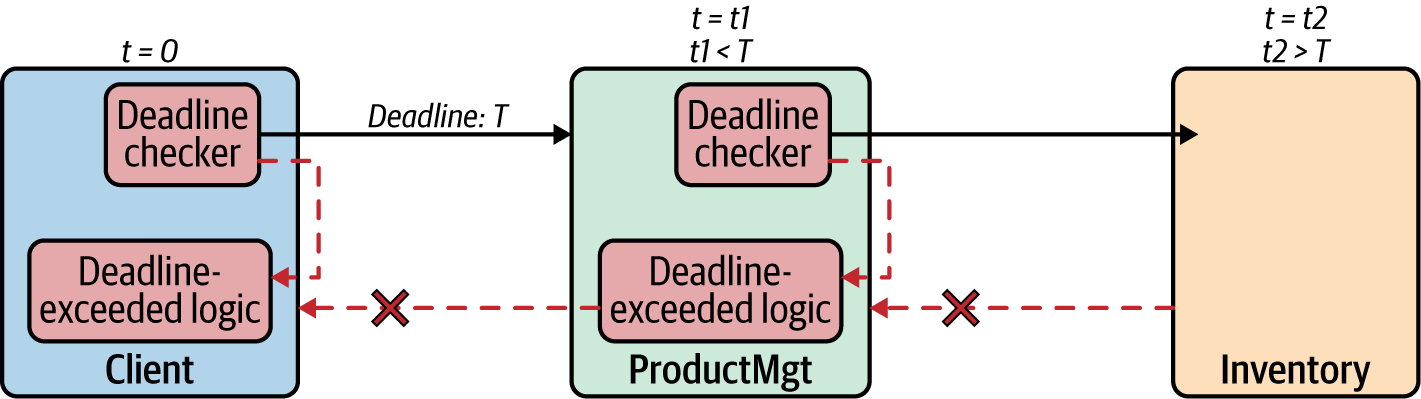
**Figure 3-10. Retry logic trying to invoke the same service a specified number of times at given intervals**

In addition to the retry logic, we may also define logic that we want to execute if the maximum retry count is reached for a given service invocation.

**Deadlines**

Deadlines are another resilient connectivity technique, similar to time-outs. With time-outs, you define a duration of time that the resilient communication logic of a given service should wait. With *deadlines,* you specify a fixed point in time that a given invocation should complete (for example, at 7:50 p.m. on...). The deadline technique is useful when you have a chain of services that a given request goes through.

In [Figure 3-11](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#deadlines), the client calls the ProductMgt service, which calls the Inventory service. At the time the client initiates the request, it can set a deadline for each request, and that deadline is propagated across all downstream services. As part of the resilient connectivity logic, each service checks the deadline of a given message. If it has expired, the service invokes the deadline-exceeded logic.



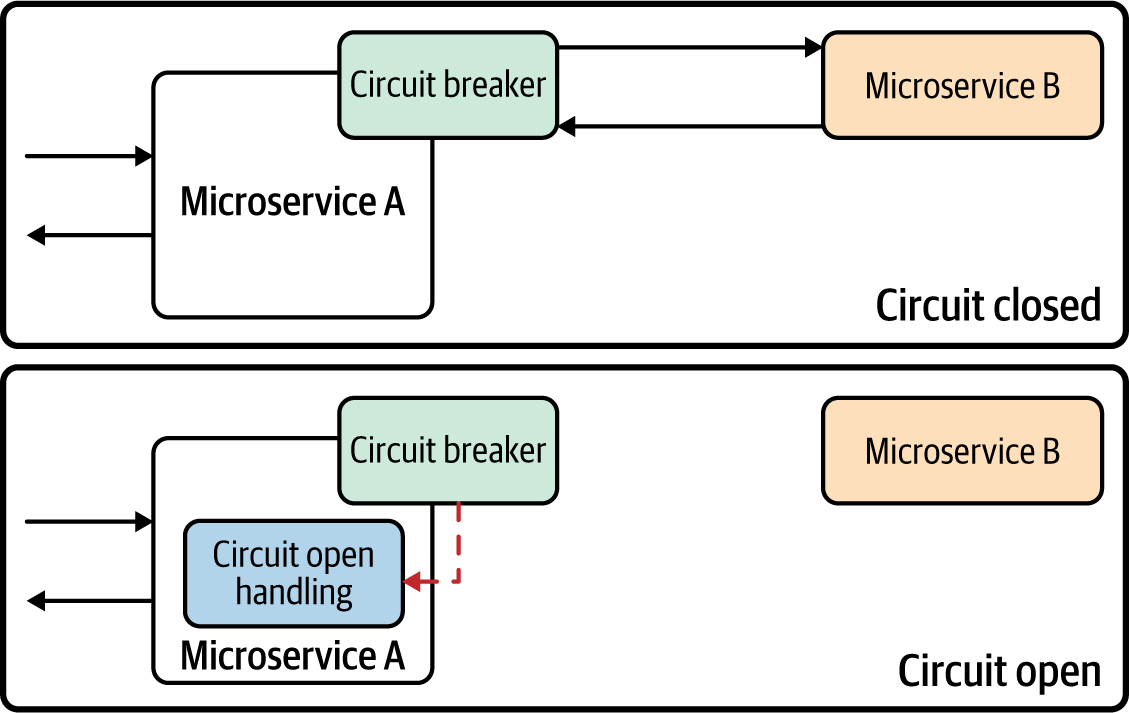
**Figure 3-11. Deadlines**

The deadline checker at each service is responsible for deadline validation as well as the propagation of deadline data across the board. When you are setting a deadline from the requester side, it should be included in predefined metadata of a request/message (which can be a message header or part of the message payload).

**Circuit breaker**

When invoking other services or systems, if the target service keeps on failing, further invocation of that service may cause more damage and cascading failures. To handle these scenarios, we can introduce a *circuit breaker* for the resilient connectivity logic of the caller microservice. A circuit breaker will prevent any further invocation of a target service if the previous service invocations have failed and the circuit state reaches a certain threshold.

Under normal circumstances, the circuit is in the *closed* state, and the invocation of the microservice takes place without any issue. However, when failures occur that match the circuit-breaker opening criteria, the circuit goes to an *open* state, preventing the invocation. [Figure 3-12](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#circuit_breaker) illustrates this process.



**Figure 3-12. Circuit breaker**

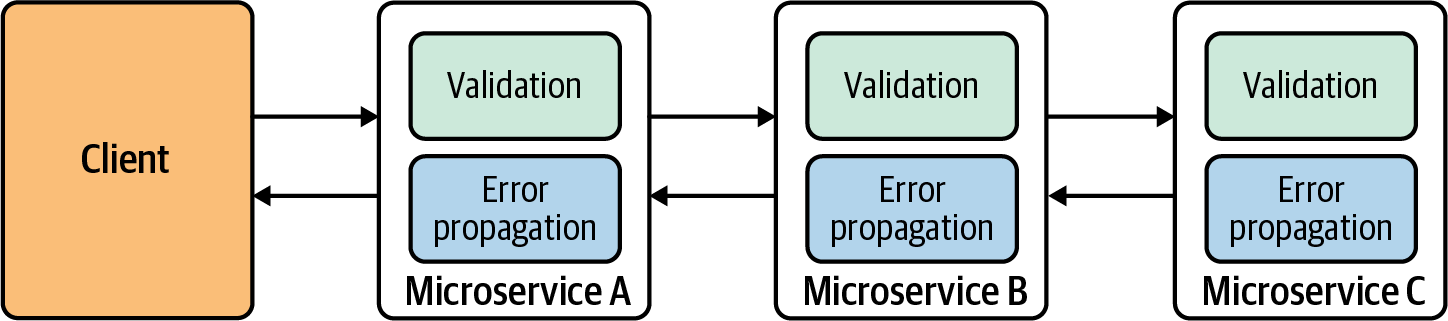
As part of the circuit-breaker configuration, we can specify multiple parameters to change its behavior. When an invocation failure occurs, the circuit breaker maintains the closed state and updates the *threshold count.* Based on the threshold count, or frequency of the failure count, it opens the circuit. When the circuit is open, the real invocation of the external service is prevented, and the circuit breaker generates and returns an error immediately without invoking the target service.

When the circuit is in an open state for a certain time period, we can apply a self-resetting behavior by trying the service invocation again after a suitable interval and resetting the breaker should it succeed. This time interval is known as the *circuit reset time-out*. When this time-out is reached, we usually say the circuit is in a *half-open state*, in which the circuit breaker allows one or more invocations of the external service as a trial. The circuit breaker changes the state to closed again if the trial succeeds, or it changes the state to open if the trial fails.

The circuit breaker is a mechanism for degrading the performance of a system when it is not operating as expected. This prevents any further damage to the system or cascading failures. We can configure the circuit breaker with the various back-off mechanisms, time-outs, reset intervals, error codes that trigger open states, error codes that trigger an ignore response, and so on.

**Fail-fast**

In distributed computing, a fast failure response is considered much better than a slow failure response. The key idea behind *fail-fast* is to detect any failures or anomalies related to service connectivity as quickly as possible. The resilient communication logic can be implemented in such a way that we validate the request prior to sending it to the target service so that we can detect any failures without even invoking the target service or system (see [Figure 3-13](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#validation_of_request_to_achieve_fail_f)).



**Figure 3-13. Validation of request to achieve fail-fast services**

We can detect failures in many ways, and they may change from one use case to another. In some use cases, we can detect failures just by looking at the message content. We can also check system resources—including thread pools, connections, socket limits, and databases—and the state of the downstream components of the request life cycle.

In most cloud native applications, resilient connectivity styles can be used either individually or as a combination (for example, time-out and circuit breaker together).

**Considerations**

Resilient connectivity is essential in building cloud native applications. We may opt to implement it as part of the service’s business logic (using libraries designed for building resilient connectivity) or as a separate runtime (known as a *sidecar,* which we’ll discuss next), or the underlying cloud service can provide these capabilities out of the box (we just need to configure them). These resilient connectivity styles can be used together with various communication patterns, such as synchronous or asynchronous, that we explored in [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns).

**Related patterns**

The Resilient Connectivity pattern is often used along with the Sidecar and Service Mesh patterns that we explore later in this chapter. Most of the techniques that we’ve discussed can be used in concert with most of the patterns related to interservice communication.

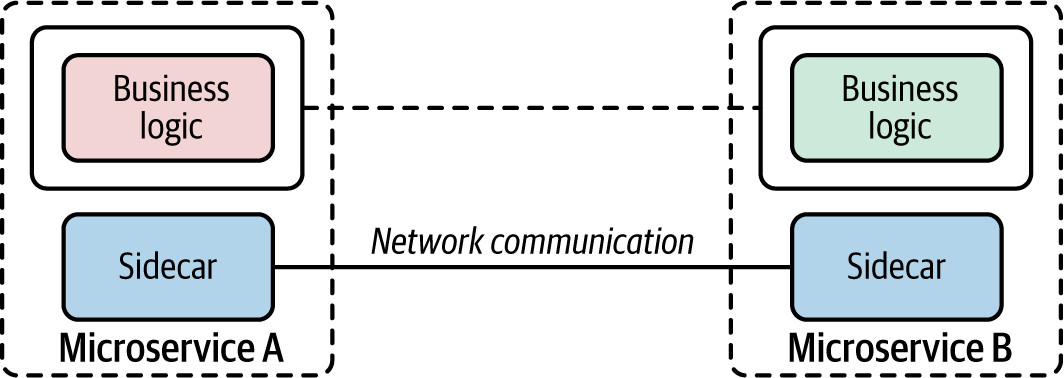
**Sidecar Pattern**

The *Sidecar pattern* is a generic pattern in which you run a colocated container (application or microservice) along with your main microservice. Sidecar containers extend and enhance capabilities of the main container. In the context of service connectivity, the Sidecar pattern is often used to implement the interservice and intersystem connectivity logic outside your main microservice.

**How it works**

Let’s look at how the Sidecar pattern is used in the context of service connectivity. Suppose we have two microservices; Microservice A and B, and we need to establish interservice communication between the two ([Figure 3-14](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#using_a_sidecar_to_offload_interservice)). The typical approach would be to build the interservice communication along with the business logic of the service. If the communication logic requires complex network communication that is independent of the business capabilities of the service, the service developers may have to spend considerable time implementing that inside the service. And when we build multiple microservices, we may have to duplicate the same capability over and over again and with multiple programming languages or frameworks.

The Sidecar pattern allows you to offload the interservice communication logic to a separate runtime that is colocated with the main microservice. When we use containers to deploy cloud native applications, a sidecar is often implemented as a colocated container with the main container that runs the business logic.



**Figure 3-14. Using a sidecar to offload interservice communication**

When you use a sidecar for interservice communication, the business logic of each service doesn’t need to worry about the underlying network communication, as the sidecar is already providing it. As the developer, you have to plug the sidecar into your main container and configure the sidecar to achieve the preferred interservice communication logic. The main container calls the colocated sidecar component (for example, via localhost), and the sidecar takes care of the external-facing communication. With this approach, no microservice should talk to other services or systems directly.

The runtime that we choose to use as the sidecar should support all the commodity features (such as secured communication, traffic routing, and service discovery) that are required for interservice communication. And we should be able to configure the sidecar by using a high-level configuration language (for example, YAML or JSON). The sidecar and main microservice share the same life cycle. It’s important to keep in mind that a sidecar can be used for any purpose that enhances the capability of the main microservice. But in this section, we focus on the service connectivity aspect of sidecar.

**How it’s used in practice**

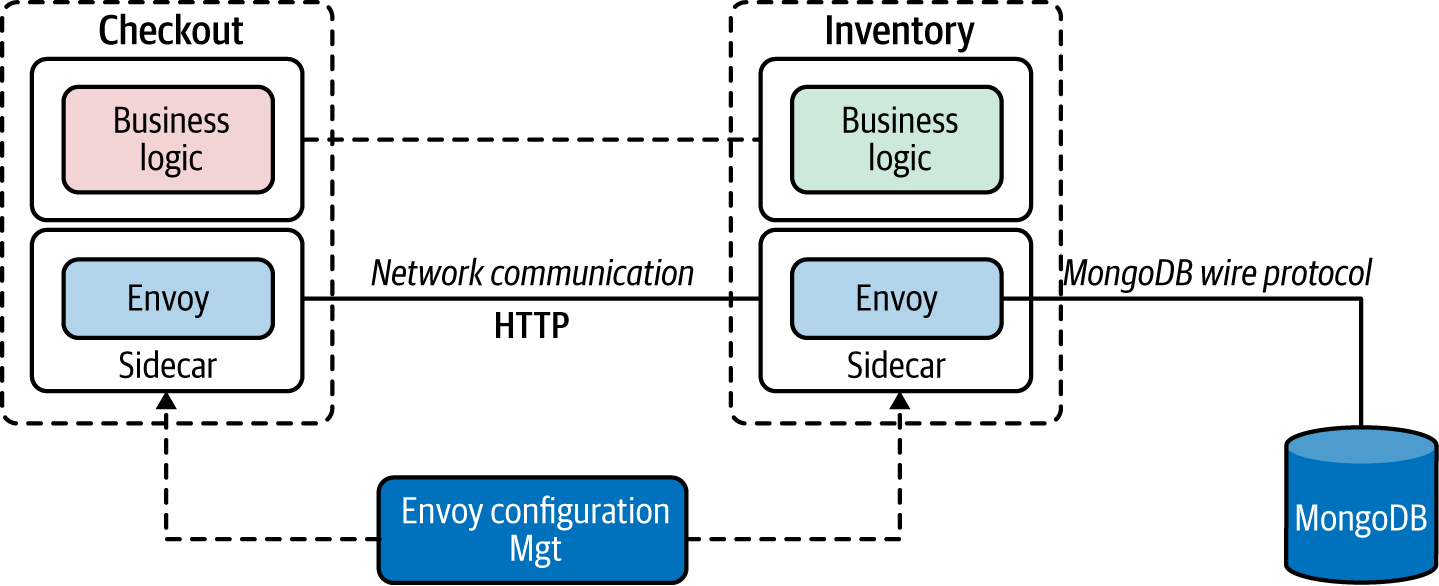
In general, the Sidecar pattern is often realized with the use of a platform such as Kubernetes, mainly because we can easily encapsulate a sidecar-based microservices application as a Kubernetes pod. So, it’s a multicontainer pod—one with the main container and the other with the sidecar. We can manage and scale the entire application as a single unit, while we have clear separation of concerns between the business logic and the extension or enhancement logic.

The Sidecar pattern is used in multiple ways when we build the connectivity between microservices or other external systems.

**Sidecar proxy**

We can use the sidecar as a *proxy* to mediate the inbound and outbound communication to the main microservices that the sidecar is attached to. Since the sidecar is being used as a proxy, the main container calls the sidecar that runs on localhost as it invokes the external service or system. Then the sidecar proxy requests the additional network communication feature, such as security or service discovery logic.

[Figure 3-15](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#using_envoy_as_a_sidecar) shows an application of this approach in a typical cloud native application. Two microservices—Checkout and Inventory—communicate with each other using [Envoy](https://www.envoyproxy.io/) as the sidecar proxy. Communication between the two services takes place over the Envoy proxy through HTTP. The inventory service also uses Envoy to connect to the MongoDB database. In this case, the Inventory service uses the MongoDB wire protocol (over TCP/IP) to connect to Envoy proxies that request to MongoDB.



**Figure 3-15. Using Envoy as a sidecar**

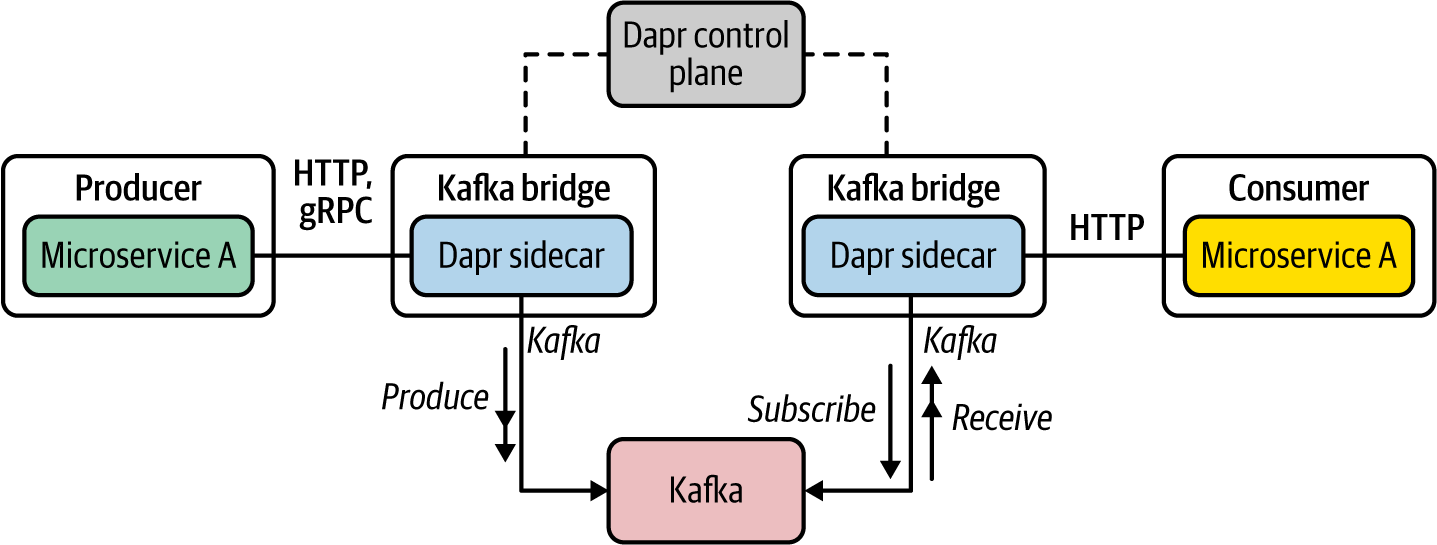
The sidecars can be configured using Envoy’s configuration management APIs to change the behavior of the sidecar proxy. For example, the communication between the two services that takes place via the Envoy proxy can be made secure and reliable with the use of Envoy features. Similarly, we can externally monitor the communication between the Inventory service and MongoDB by allowing Envoy to sniff the communication (by using Envoy’s MongoDB sniffing filters) and report to an external monitoring tool.

**Sidecar bridge**

In the sidecar proxy approach, the proxy didn’t alter the inbound and outbound protocols. It simply connected the main container to the external services and systems using the same protocol. However in the sidecar bridge approach, we use the sidecar to bridge two different protocols.

For example, suppose your main container wants to communicate with only HTTP and still wants to connect with messaging systems such as Kafka. The main container and sidecar use completely different protocols as well as completely different messaging patterns. We can use a sidecar bridge to achieve communication between the two.

The example in [Figure 3-16](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#using_a_dapr_sidecar_as_an_http_kafka_b) uses [Dapr](https://dapr.io/) as the sidecar. Microservice A calls the sidecar APIs through HTTP to produce messages to Kafka. The sidecar is configured to connect with the Kafka service, which is transparent to the microservices. On the consumer side, Microservice B registers (using HTTP API) with the Dapr sidecar to receive messages that are published to a given topic. The Dapr sidecar then subscribes to that topic in Kafka by using the Kafka protocol. So, when there’s a new message for that topic in Kafka, the Dapr sidecar receives the message and then forwards it to Microservice B via HTTP.



**Figure 3-16. Using a Dapr sidecar as an HTTP-Kafka bridge**

As you can see, the use of a protocol-bridging sidecar vastly simplifies the business logic of the microservice. As a developer, you can now focus on the microservices’ business logic while the sidecar takes care of connecting the microservice to a wide range of systems and services with minimal effort.

**Considerations**

The Sidecar pattern is one of the most popular patterns used in cloud native application development. The capabilities that it brings can greatly enhance and extend the main container. However, keep in mind that using this pattern comes with a price. Here are some of the key considerations to be aware of when using this pattern:

* Using a sidecar along with a microservice multiplies the number of instances you need to manage and run. If you have four microservice instances, then with a sidecar you need to run eight instances.
* Management of sidecar containers needs to be done via a dedicated control plane component. While it is possible to invoke the configuration API via standard protocols such as HTTP or gRPC, it is more efficient and easier to manage sidecars via a dedicated control plane component.
* Sidecar configuration can rapidly grow to complex logic. The more services and systems that you connect with, the more complex the sidecar configuration that you need to manage.
* Never implement any business-logic-related capability inside the sidecar. That would violate the key purpose of the Sidecar pattern; the leaking of business logic to multiple layers also may have adverse consequences such as management and ownership nightmares.

With the overwhelming success and usage of the Sidecar pattern, some platforms (such as Kubernetes) are planning to support sidecars as a first-class construct in the platform in the future.

**Related patterns**

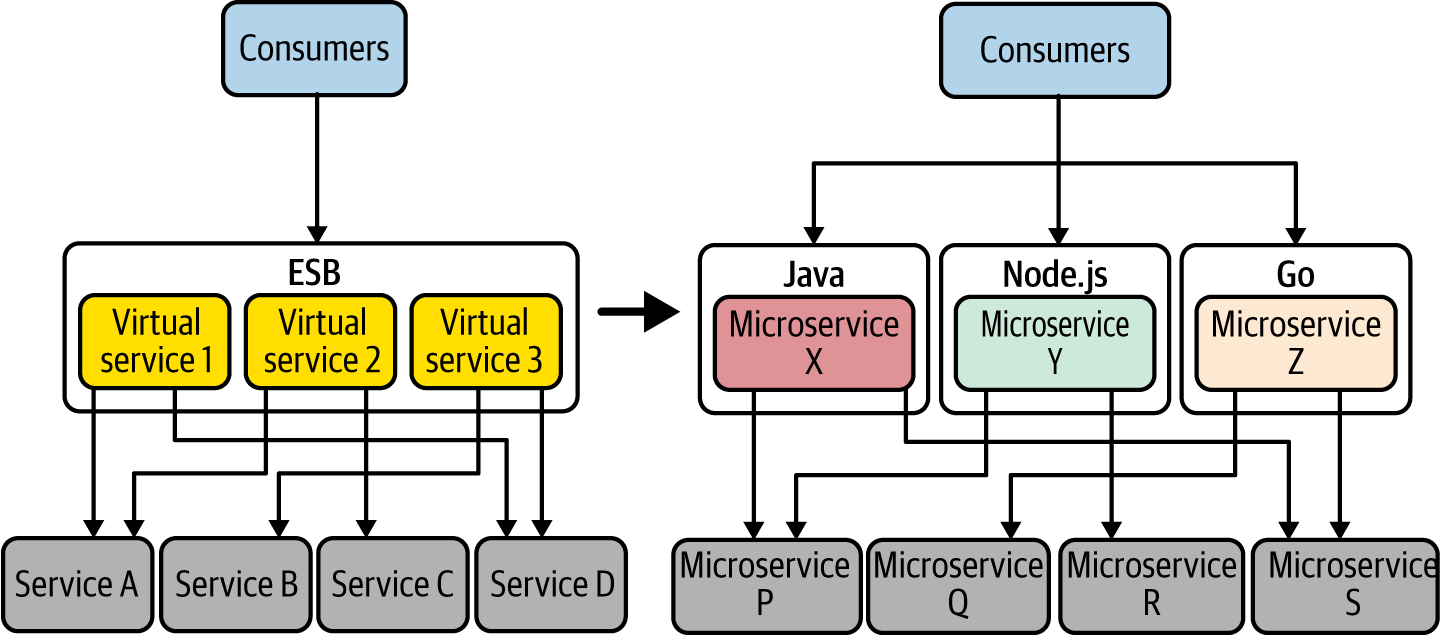
The Sidecar pattern is closely related to, and extended, when implementing the Service Mesh pattern, which we discuss next.

**Service Mesh Pattern**

The *Service Mesh pattern* is essentially an extension of the Sidecar pattern, to be used as the communication infrastructure of a cloud native application.

The main motivations behind the Service Mesh pattern are the challenges that we started to encounter when building the connectivity between microservices and systems of a cloud native application. As we discussed in [Chapter 1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#introduction_to_cloud_native), we used to use the centralized ESB architecture to connect services and systems. With the elimination of ESB, now the microservices themselves need to take care of the interservice communication logic.

[Figure 3-17](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#interservice_communication_from_esb_lef) depicts the two architectures: ESB on the left, and microservices on the right. Say you need to call multiple downstream services in a resilient manner (including time-outs and retries) and expose the functionality of another microservice. With the ESB architecture (left), you can use the built-in capabilities of ESB for building reliable communication with minimal effort. However, when you use microservices architecture (right), the interservice communication logic must be part of the microservices you build. Your microservice code needs to take care of both the business logic and the connectivity logic.



**Figure 3-17. Interservice communication: from ESB (left) to microservices (right)**

Building this network communication logic as part of your microservices drastically complicates the business logic and increases the development time for all your microservices. You will have to rely on external libraries (for example, Resilience4j) to build these interservice communication features. Also, if you use multiple technologies or programming languages, you will have to duplicate the effort across multiple technology stacks (for example, the circuit breaker has to be implemented in Java, Node.js, or Go).

Since most of the interservice communication requirements are generic across all microservice implementations, we can think about offloading all such tasks to a different layer such as the sidecar, so we can keep the service code independent. This is the key idea behind the Service Mesh pattern.

**How it works**

The Service Mesh pattern allows you to have an interservice communication infrastructure between your microservices and other systems. With a service mesh, a given microservice won’t directly communicate with the other microservices. Rather, all service-to-service communications take place through a sidecar proxy. As illustrated in [Figure 3-18](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_mesh_components), the Service Mesh pattern introduces the following components to provide a simple, scalable, and configurable communication infrastructure:

*Service Mesh sidecar proxy*

This is known as the *data plane,* in which all the interservice communication logic is applied to the messages exchanged between services and systems.

*Control plane*

Sidecar proxies are controlled through the control plane. This centralized component provides a rich and simple API to control sidecar proxies of the data plane.

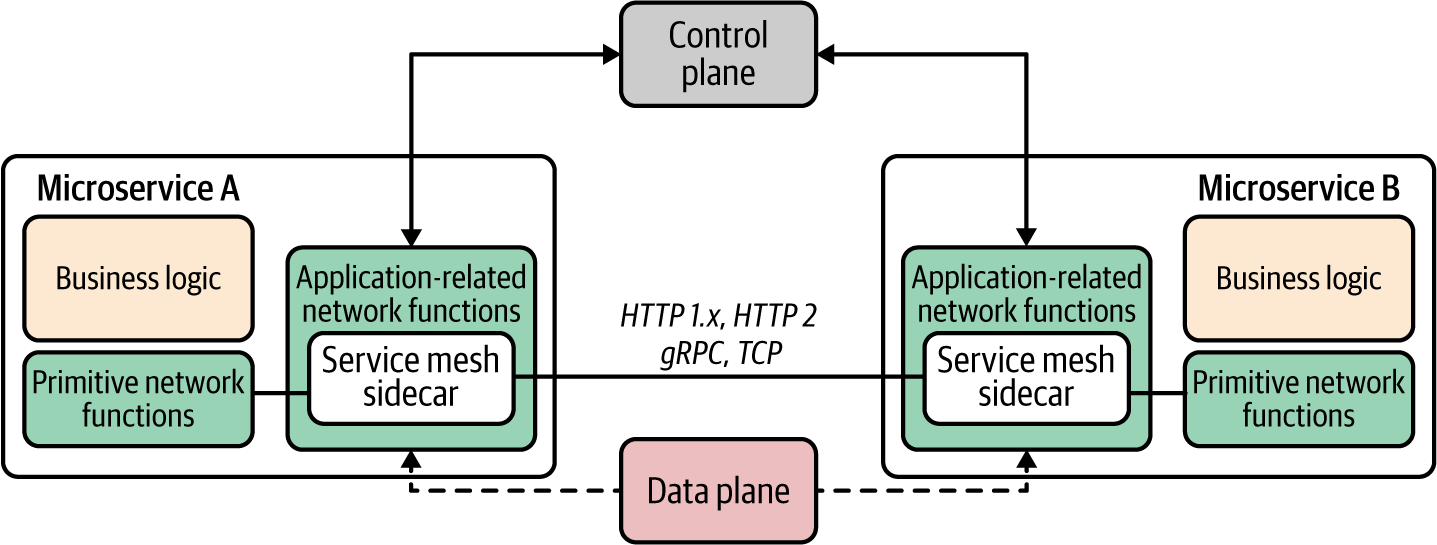
*Service Mesh configuration language*

This is the configuration API that allows you to configure the data plane to control the interservice communication logic.

*Built-in support*

This support provides reliability, security, observability, service discovery, policy enforcement, and more.

These components work together in building interservice connectivity in our cloud native applications. As the service developer, you build your microservices and deploy them along with a sidecar. This process, known as *sidecar injection*, can be done manually or can be automatically injected during deployment. As we discussed in the Sidecar pattern, the business logic communicates with the sidecar via localhost communication (which is denoted as Primitive Network Functions in [Figure 3-18](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_mesh_components)). Then the sidecar takes care of connecting with any external system. Both the sidecar and main container are deployed as a single unit (for example, in a Kubernetes pod).



**Figure 3-18. Service Mesh components**

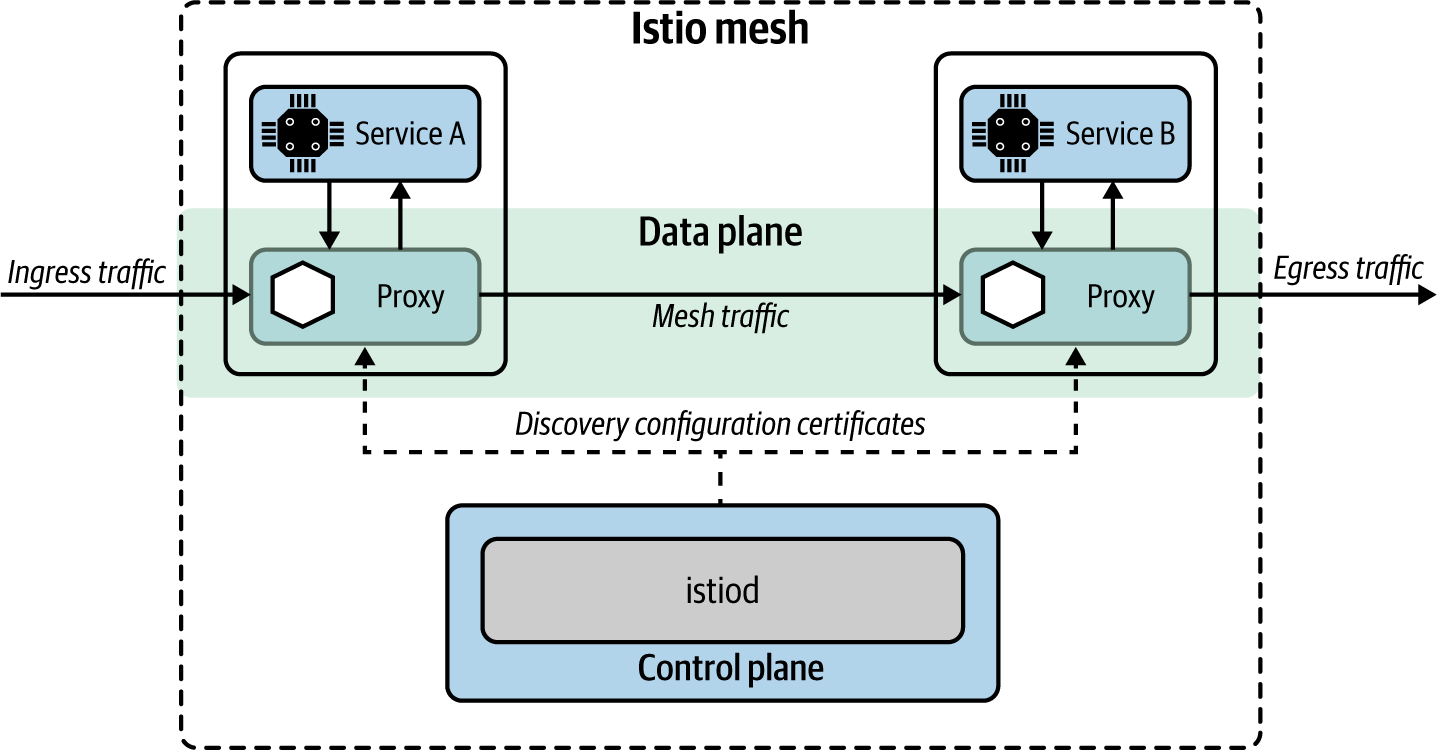
Service Mesh implementations define a configuration language or API to control the various capabilities and control the data plane via that configuration. The control plane connects all sidecars to a central location and enables the developers to manage their services that run on top of the service mesh. The capabilities such as reliability, security, observability, service discovery, and policy enforcement are applied at the data-plane level while they are controlled through the control plane.

**How it’s used in practice**

The Service Mesh pattern is used in complex microservices deployments where we need to manage an increasingly large number of microservices. Often the service mesh is built on top of a container orchestration layer such as Kubernetes to reduce the overhead of managing containers and to use the abstractions provided by Kubernetes such as services and pods. Quite a few service mesh implementations are available: Istio and Linkerd are the most popular. Each service mesh has its own configuration language and API.

[Figure 3-19](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#component_architecture_of_istio_service) shows the component architecture of Istio. For any microservice that you develop, you can enable Istio sidecar injection so that sidecar proxies can intercept all network communication between microservices and other systems. Then, using the Istio control plane, you can configure and manage the communications.

The control plane consists of *Istiod* components that provide service discovery, configuration, and certificate management. The sidecar proxies are managed by Istiod, and users control the mesh behavior via Istiod.



**Figure 3-19. Component architecture of Istio service mesh**

Key features of Istio include the following:

*Automatic load balancing*

For HTTP, gRPC, WebSocket, and TCP traffic

*Traffic control*

Includes routing rules, retries, failovers, and fault injection

*Policy enforcement*

A pluggable policy layer and configuration API supporting access controls, rate limits, and quotas

*Observability*

Metrics, logs, and traces for all traffic within a cluster

*Security*

Service-to-service communication in a cluster with strong identity-based authentication and authorization

In addition to standalone service mesh offerings, cloud vendors such as Google Cloud offer Istio as a managed service.

**Considerations**

Although the Service Mesh pattern is a popular concept these days, adopting it in the real world to build cloud native applications should be done with caution. Here’s why:

* Managing a service mesh deployment can be overwhelmingly complex. The complexity comes from the sidecar architecture (in which we need to run one extra container for each service instance) as well as from the architecture of the service mesh implementation (we need to manage multiple service mesh components that interact with each other).
* A service mesh is often built on top of containers and container orchestration platforms such as Kubernetes. This may double the complexity that it brings in.
* Running and managing a fleet of sidecar proxies carries a major performance overhead.
* Service Mesh doesn’t offer first-class support for asynchronous event-driven communication yet.

Service Mesh as a service offering is quite pragmatic, and you will be able to overcome most of the operational complexity of managing it yourself.

**Related patterns**

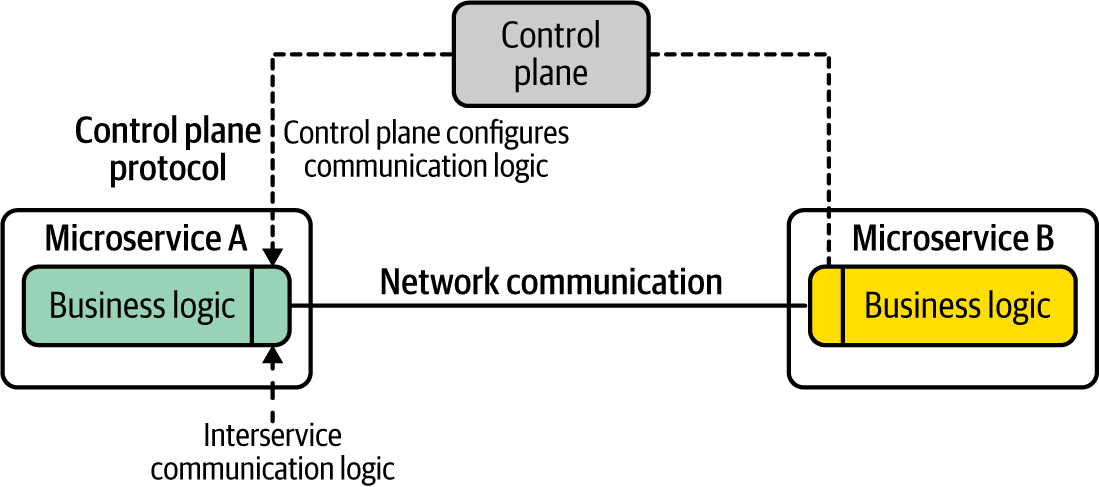
The Service Mesh pattern is closely related to the Service Connectivity and Sidecar patterns (described in this chapter).

**Sidecarless Service Mesh Pattern**

As you have seen in the previous section, the service mesh requires a fleet of sidecar proxies for each microservice instance that you run in your cloud native application. This is one reason for the slow adoption of the Service Mesh pattern. The *Sidecarless Service Mesh pattern* tries to solve that problem by eliminating the need for a sidecar. The application of this pattern is still in its early stages, but because of its unique advantages, it’s quite a promising pattern in the context of microservices connectivity.

**How it works**

The key idea behind the Sidecarless Service Mesh pattern is this: if the control plane can manage and control the network communication of the sidecar proxy, why not directly do it with the client component of the main container? Suppose two microservices need to communicate with each other, as shown in [Figure 3-20](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#sidecarless_service_mesh-id00201). Similar to the Service Mesh pattern, we can use a control plane to manage and configure the communication (mesh traffic) between microservices. Rather than using a dedicated sidecar proxy to handle the interservice communication, we can embed the sidecar proxy logic to the microservice runtime itself. For example, Microservice A’s runtime contains the business logic as well as the logic related to the mesh traffic. The embedded runtime understands the control plane configuration commands that come through using a control plane communication protocol.



**Figure 3-20. Sidecarless service mesh**

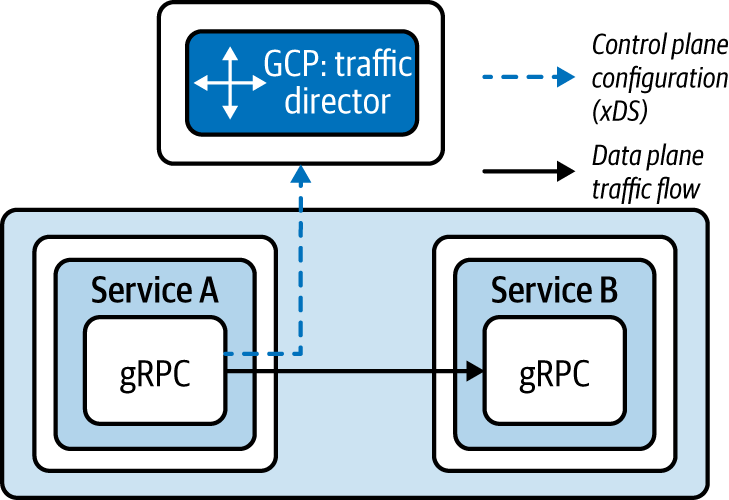
As you can see in the figure, the control plane defines the configuration API and the control plane protocol. The technology that we use to build both microservices implements a client application that understands the control plane configuration protocol. For example, in this use case, suppose Microservice A is using a Java client to build the interservice communication. If we build the support to control that client component via the control plane configuration protocol, all the interservice communication capabilities can be applied at the microservice level itself (same runtime) while being centrally managed via the control plane. However, this requires each implementation technology to support the control plane API and apply the network communication logic at each network communication layer.

**How it’s used in practice**

The Sidecarless Service Mesh pattern can be used to implement various aspects of microservices communication. It can be used to implement a full-blown service mesh or to implement a selected set of features offered from a service mesh.

**Sidecarless gRPC services in Google Traffic Director**

One of the first implementations of the Sidecarless Service Mesh pattern was realized with Google Cloud’s Traffic Director. As shown in [Figure 3-21](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#sidecarless_service_mesh-id00202), Traffic Director is the control plane that manages microservices-based applications running in the system and can control Envoy sidecar proxies via Envoy’s configuration API xDS. In the sidecarless deployment of gRPC-based microservices, the gRPC client side implements Envoy’s configuration protocol xDS so that the control plane can control the communication flows through the gRPC client.



**Figure 3-21. Sidecarless Service Mesh**

Traffic Director gives the gRPC clients information about which services to contact, how to load-balance requests to multiple instances of a server, and what to do with requests if a server is not running. As long as you use a gRPC client that implements Envoy xDS API, you no longer need to have a sidecar proxy to bring the service mesh capabilities to your gRPC applications. In this case, the gRPC client implements only a selected set of features of a service mesh, but we can further improve it to support all its other features.

When it comes to the development experience of microservices that you deploy on top of a sidecarless service mesh, an implementation of a control plane configuration API is transparent to the service developers. It is implemented in the framework’s or client library’s network communication logic.

**Considerations**

Sidecarless Service Mesh is an emerging pattern that tries to overcome the limitations of the conventional sidecar-based service mesh architecture. The main complexity it introduces is that the libraries that we are using need to have support for the service mesh control plane APIs and implement the network communication logic as part of that client library itself.

For example, suppose you want to use the circuit-breaking capability when you call an external service. The client library that you use to build the microservices that invoke the other service then needs to offer support for the xDS protocol for the control plane, as well as implement the actual circuit-breaking logic as part of the client library. Therefore, the adoption of this pattern largely depends on availability of such network communication libraries for a wide range of programming languages.

**Related patterns**

Sidecarless Service Mesh is an alternative to the Service Mesh pattern with a sidecar proxy. The patterns related to resilient communication are often implemented at the control plane–compliant client libraries that we use to build the microservices.

**Technologies for Implementing Service Connectivity Patterns**

Let’s discuss some of the technologies that you can use to implement the connectivity patterns in this section. For connectivity of cloud native applications, platforms such as Kubernetes offer most of the capabilities required for patterns such as Service Abstraction, Service Registry and Discovery, and Sidecar. Similarly, these features are built-in capabilities of cloud services such as AWS, Azure, and GCP.

Kubernetes primarily drives the use of the Service Abstraction pattern in the context of container and container orchestration. However, many cloud services use the notion of a service in most of their offerings. The scope and capabilities of the service may be dramatically different based on the context and use case. For example, serverless platforms such a [Knative](https://knative.dev/) use service abstraction to deploy an application, while service mesh solutions such as Istio use a [virtual service](https://oreil.ly/uqnMz) abstraction, which is a unit of application behavior bound to a unique name in a service registry. An Istio service consists of multiple network endpoints implemented by workload instances running on pods, containers, and VMs. Therefore, we need to choose the implementation technology by looking at the actual use case for which we need service abstraction.

The service mesh solutions offer built-in support for resilient communication, security, observability, service discovery, and traffic routing. The service mesh technology space is growing rapidly, but Istio and Linkerd are the most popular implementations out there. The production-level adoption of service mesh is still low because of the complexity of managing it and resource consumption due to having a sidecar per each service instance. Some cloud vendors such as GCP offer managed service mesh offerings, which makes life easier for users.

If you are not using a service mesh, or the underlying platform (cloud services) doesn’t support its features (such as resilient communication or service discovery), then those features need to be implemented at the microservice level by using dedicated client libraries. You need to choose a library that enables resilient communication for the service development technology you’re using. Several libraries are available for various programming languages (for example, we can use [Reslience4j](https://oreil.ly/KgIvr), [Quarkus](https://oreil.ly/dNdm2), or [Micronaut](https://oreil.ly/tPonw) for Java, and frameworks such as [Go kit](https://oreil.ly/ZZYpw) for Go).

If you plan to use the sidecar architecture to implement certain connectivity patterns, you can use [Envoy](https://www.envoyproxy.io/), which supports a wide range of them. Also, projects such as [Dapr](https://github.com/dapr/dapr) offer higher-level abstractions related to connectivity such as sidecar bridges. Sidecarless architectures are also at a very primitive stage, and cloud vendors such as Google Cloud support sidecarless architecture for a selected set of protocols such as gRPC. The control plane configuration protocols such as Envoy’s [xDS](https://oreil.ly/pQ18U) play a vital role in the success of the sidecarless architecture.

**Summary of Connectivity Patterns**

[Table 3-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_pattern) lists the connectivity patterns, and details when and when not to use them.

| **Pattern** | **When to use** | **When not to use** |
| --- | --- | --- |
| Service Connectivity | This is a generic pattern that you can use to build connectivity in almost all the cloud native applications. | (Not applicable.) |
| Service Abstraction | Usually, you need to explicitly use it if you are using Kubernetes or a cloud service. Useful when you connect cloud native applications with existing monolithic systems. | Not required to specifically use this pattern when you are fully dependent on a cloud service or a serverless platform. |
| Service Registry and Discovery | A fully fledged service registry and discovery solution is required if you have several dozen services consumed by a wide range of clients across the organization and beyond. For most use cases, the foundational service registry and discovery offered from platforms such as Kubernetes should be sufficient. If you use a cloud service such as AWS, Azure, or GCP, most of the capabilities are available out of the box. | If the number of services that you need to connect is small, having a full-blown service registry and discovery service doesn’t make sense. You will still need a primitive service discovery mechanism (for example, DNS) to encapsulate service location and deployment details. |
| Service Resilience | Often required when building a reliable cloud native application that connects with multiple services and systems. Essential for connecting legacy systems with cloud native applications. Explicitly implement resilience if the underlying cloud service or deployment (for example, service mesh) doesn’t support resilient connectivity. | Not required to explicitly use if you are building the application on top of a service mesh, cloud service, or using a serverless platform (they offer out-of-the-box support for resilient connectivity for the most part). |
| Sidecar | Useful when you have to decouple the business logic from the connectivity logic. If the connectivity logic is too complex, offloading it to a separate runtime makes sense. You use polyglot technologies that require the same connectivity features. | Not suitable if your DevOps don’t have the capacity to handle the complexity of sidecar architecture. If you don’t use container orchestration, it’s overwhelmingly complex to support sidecar architecture. |
| Service Mesh | You have to connect numerous microservices to achieve resilience, traffic routing, secured communication, service discovery, and observability. | (Same as Sidecar pattern.) |
| Sidecarless Service Mesh | Useful if the sidecar architecture hinders performance. The underlying implementation technology supports sidecarless interaction with control planes. | Still at very early stages. So, it is better to avoid it unless the pattern is offered from the technology stack or cloud provider (for example, GCP Traffic Director). |
| Table 3-1. Connectivity patterns | | |

**Service Composition Patterns**

In a cloud native application, the interaction among microservices is what builds the business capabilities of that application. (For example, in our online retail application example, the Order microservice needs to interact with the Catalog microservice and the Payment microservice, and so on.) When building any business capability, you have to connect one or more microservices and other systems. In the previous section, we discussed patterns related to connecting services and systems at a more operational or infrastructure level. In this section, we focus on the patterns that you can use when realizing the business logic of a service, as well as how to create composite capabilities by using multiple services or systems.

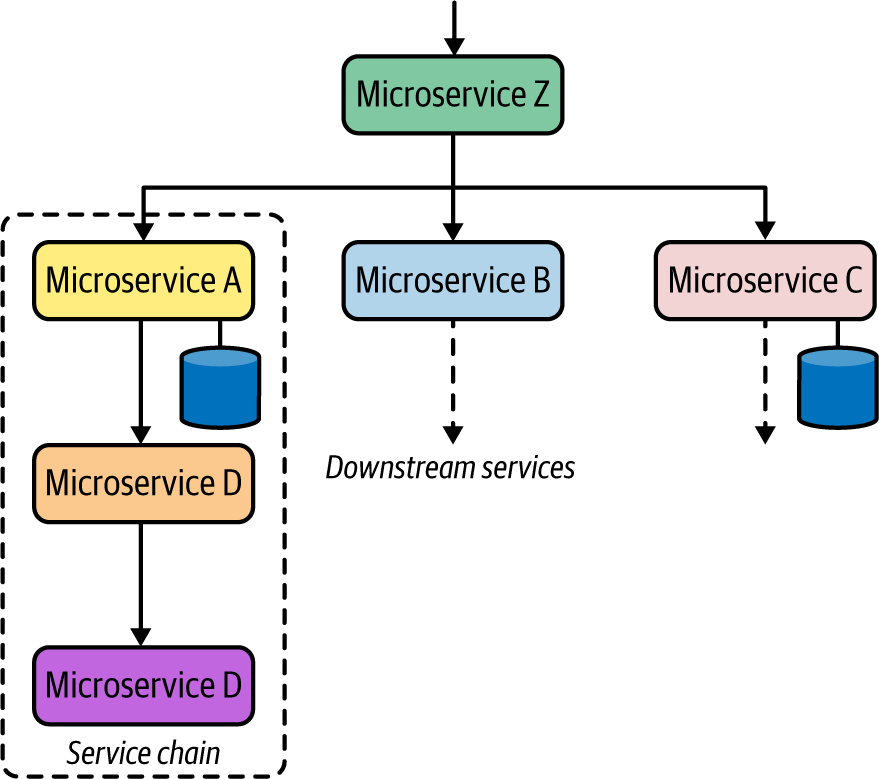
*Service composition* is all about how you implement a business use case by plumbing, or integrating, multiple services and systems. It’s important to keep in mind that the services that we build use existing services, and should have a clear business scope and be driven by clear business requirements. We cannot simply create composite services by randomly connecting them. For instance, if you want to support a certain business capability (such as order management), you should come up with the API of the composite service and the downstream services and systems that you want to integrate with. Next we’ll discuss three common service composition patterns: Service Orchestration, Service Choreography, and Saga.

**Service Orchestration Pattern**

*Service Orchestration* is a well-known composition pattern from the era of SOA. In the context of cloud native applications, when we have to build a business capability by invoking multiple services and systems, the composition logic is implemented in a single microservice. This pattern, for the most part, uses synchronous communication and operates in a stateless way.

**How it works**

Service Orchestration implements the business logic of a microservice by invoking and integrating one or more microservices and systems. Suppose you are building a new business capability implemented as Microservice Z ([Figure 3-22](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_orchestration_logic_implemented)). It requires the integration of several existing microservices such as Microservices A, B, and C. You can build the business logic of Microservice Z so that it invokes Microservices A, B, and C with the required messages and finally sends the response back to the consumer of Microservice Z. The downstream services may use disparate message formats and communication protocols. Therefore, Microservice Z needs to handle all that complexity. The entire composition logic is self-contained within Microservice Z’s scope.



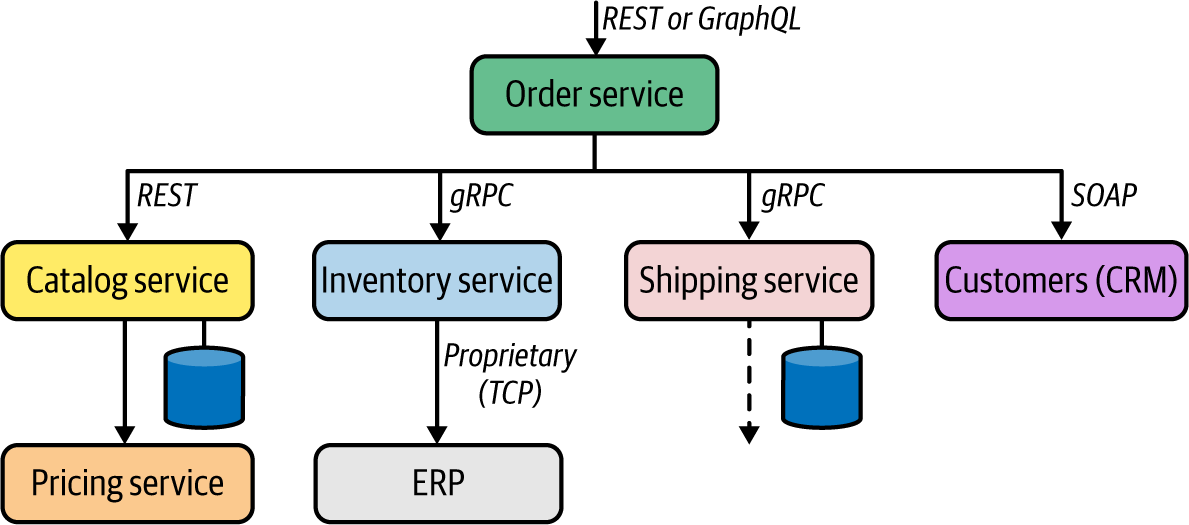
**Figure 3-22. Service Orchestration logic implemented in Microservice Z**

The downstream services can use either synchronous or asynchronous communication. These downstream services may call several other downstream services too. This style of communication is known as *service chaining*. From the perspective of composite services such as Microservice Z, the existence of multiple chained downstream services is irrelevant.

**How it’s used in practice**

Service Orchestration is commonly used in most cloud native applications, as microservices need to integrate with one or more other microservices or systems when implementing a given business capability. As discussed earlier, before the cloud native era, we used to use technologies such as ESBs or workflow engines to build this integration logic. In the context of cloud native applications, we rarely use conventional monolithic solutions such as ESBs or workflow engines to build orchestration logic. Instead, we build a microservice that orchestrates the composition logic.

[Figure 3-23](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_orchestration_in_a_real_world_s) shows a typical service orchestration scenario within a real-world online retail application. The Order service’s business logic requires the orchestration of the calls to four microservices: Catalog, Inventory, Shipping, and Customers (CRM). As you can see, these service invocations are using disparate communication protocols such as REST over HTTP, gRPC, and SOAP. When we are building this kind of microservice, the orchestration logic is implemented as part of the Order service’s business logic.



**Figure 3-23. Service Orchestration in a real-world service composition use case**

When making technology choices to implement such a service, we can use a generic programming language (such as Java, Go, or C#) or a cloud native integration framework such as Apache Camel that has a lot of abstractions to simplify the service integration.

**Considerations**

When using Service Orchestration, we need to be aware of several considerations:

* Use this pattern if it can be directly mapped to a business capability that aggregates the capabilities of several other downstream capabilities. Otherwise, you will be creating a monolithic service with multiple business capabilities.
* Service Orchestration is straightforward to implement in scenarios that are stateless, so we don’t need to worry about preserving the state of the orchestration.
* It is better to limit the number of service calls in a composition. For instance, if you have to orchestrate calls among more than four or five services, that’s a sign of business scope issues with the service, or perhaps the downstream services may be too granular.
* Service Orchestration centralizes the composition logic to a single service, and that service is tightly coupled to all the downstream services that it connects with.

Avoid using a conventional monolithic technology such as an ESB or a workflow engine to implement the orchestration logic, as they are not designed for building cloud native applications.

The composite services that we develop by using Service Orchestration are exposed to the consumers via an API management layer, which we will explore in [Chapter 7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_and_consumption_patterns).

**Related patterns**

Service Orchestration is used alongside other service composition patterns such as Service Choreography; most use cases require a hybrid of both patterns. Most of the foundational communication patterns introduced in [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns) can be applied when building service orchestrations.

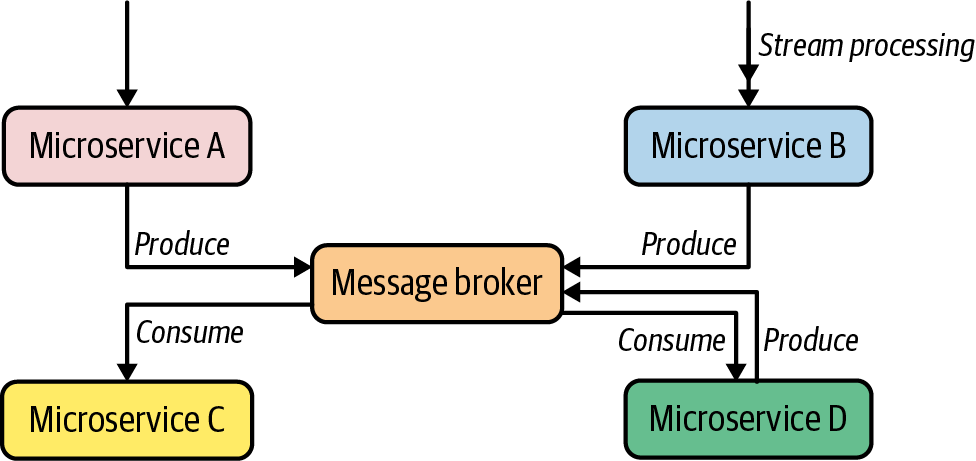
**Service Choreography Pattern**

When building service compositions, we don’t always want to centralize the composition logic to a single service. In some cases, we need to build it across multiple services. The *Service Choreography pattern* creates service compositions by using asynchronous communication between microservices and other systems.

**How it works**

At the heart of the Service Choreography pattern, we build a business use case that requires interaction among multiple microservices and other systems by creating asynchronous event-driven communication links with the use of a message broker (or event hub). The interaction logic is dispersed across multiple microservices, and no direct coupling occurs between microservices. Unlike in the Service Orchestration pattern, microservices do not actively invoke other microservices, but operate more or less in a reactive mode based on the events and messages coming into the service. Hence the microservices that we use in Service Choreography are also known as *reactive microservices.* (Some of the core concepts in Service Choreography are closely related to the event-driven architecture patterns that we explore in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns).)

Microservices interact with one another through the events coming in from, and events published to, the broker ([Figure 3-24](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_choreography_pattern-id00270)).



**Figure 3-24. Service Choreography pattern**

The composition logic is formed by publishing messages to queues or topics and the consumer microservices subscribing to them. By using asynchronous messaging patterns such as Single-Receiver (queue-based) or Multiple-Receiver (pub-sub with topics), we can create distributed asynchronous composition logic across multiple microservices.

In [Figure 3-24](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_choreography_pattern-id00270), Microservice A receives an event and publishes the resulting event to a queue in the broker. Another service consumes that message from the queue and executes the service’s business logic upon the receipt of that message. Similarly, we can publish the same event to multiple consumers via a topic. A given service operates autonomously and is responsible only for processing the event and publishing the result to the broker. Also in a Service Choreography scenario, services can process a stream of events and publish results to the broker (in this example, Microservice B). Based on the requirements, the messaging among these services can have additional reliability guarantees such as at-least-once ([Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns) covers these guarantees in detail).

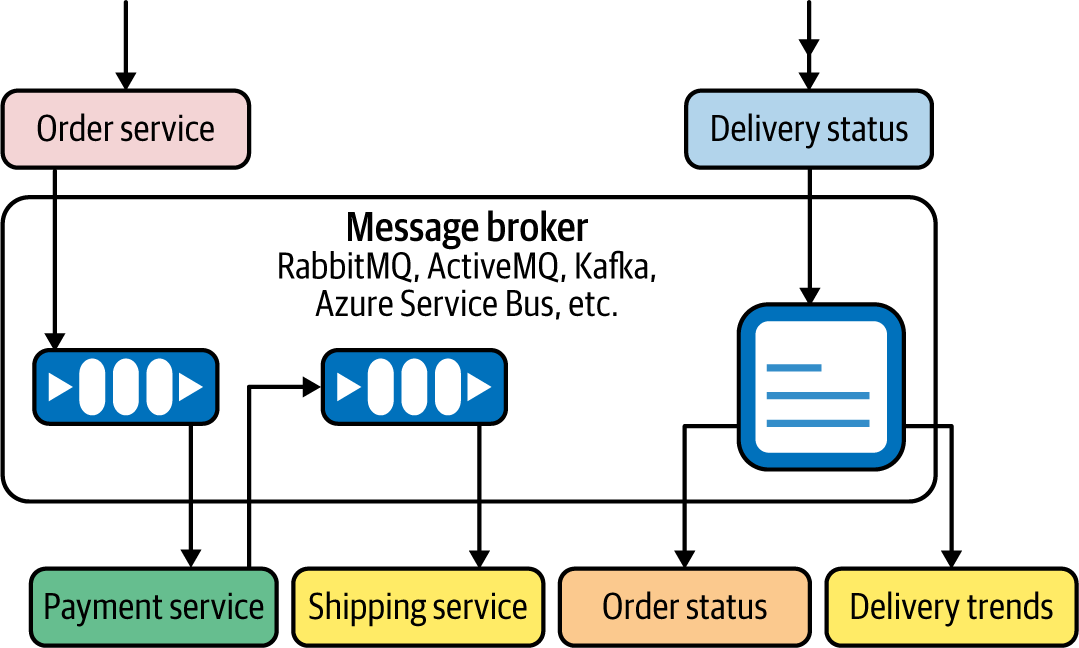
It’s important to note that in the Service Choreography pattern, we use the message broker as a *primitive messaging infrastructure.* We don’t put any business logic (such as routing based on a certain criteria) inside the broker. All the logic should reside inside either producer or consumer microservices.

**How it’s used in practice**

The realization of the Service Choreography pattern doesn’t require any specific technology or framework. It requires only the coordination between the service and system using asynchronous messaging via a message broker.

In an online retail application designed as a cloud native application, multiple services exchange events based on the business use case that we need to support ([Figure 3-25](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_choreography_in_action_with_que)). For example, the Order service gets an order-place event and enqueues it in the broker. Then the Payment service that listens to that queue processes the order and verifies the payment. The result of the payment verification is also sent to another queue, which the Shipping service listens to. When that event reaches the Shipping service, it knows that the payment is already verified, and we are good to proceed with the shipping process. Similarly, the Delivery Status service receives a stream of events, and the results are published as events to a topic in the broker. Several services may be subscribed to that topic (such as Delivery Trends and Order Status), and they will receive messages via pub-sub.

It is also possible to use multiple and disparate broker solutions to facilitate communication among services. It is not mandatory to stick to a single broker, but based on the use case, you can use multiple message broker solutions to implement different types of asynchronous communication. For instance, you can use an AMQP-based broker for implementing reliable and guaranteed delivery between services, while you can use a broker such as Kafka for pub-sub and highly scalable use cases that have fewer constraints on delivery guarantees.



**Figure 3-25. Service Choreography in action with queue- and topic-based asynchronous messaging**

**Considerations**

Some of the key considerations to be aware of when adopting the Service Choreography pattern include the following:

* Service composition logic is dispersed across multiple microservices. Unlike in Service Orchestration, you can’t understand the business logic of a choreography scenario by just looking at a single service.
* Services are loosely coupled. Adding or removing services is much easier with the Service Choreography pattern compared to Service Orchestration.
* Because of the event-driven and asynchronous nature of the Service Choreography pattern, you can implement it by using a serverless platform. For instance, you can model all the event-driven microservices as serverless functions.

**Related patterns**

Service Choreography is often used along with Service Orchestration in a hybrid way. Most use cases require both synchronous and asynchronous service interactions. When realizing Service Choreography, we use asynchronous communication patterns such as the Single-Receiver and Multiple-Receiver patterns introduced in [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns).

**Saga Pattern**

When we create service compositions by using multiple microservices, we may have to execute those service interactions in a transactional way (for example, if one service interaction fails, the rest of the service interactions should be rolled back). As the transaction boundary of such a scenario spans multiple microservices and other systems, this is known as a *distributed transaction*.

The *Saga pattern* provides a way to build distributed transactions that span multiple microservices. It does this by using corresponding compensation operations to undo every service interaction that is part of a single distributed transaction.

**NOTE**

The Saga pattern was introduced in a [paper](https://oreil.ly/MCjoW) published in 1987 by Hector Garcia-Molina and Kenneth Salem.

The Saga pattern can be applied in both the Service Orchestration and Service Choreography patterns that we discussed in previous sections.

**How it works**

The Saga pattern aims to build distributed transactions across multiple microservices and other systems by breaking a given transaction into a sequence of subtransactions and corresponding compensating transactions. All transactions in a Saga either complete successfully, or, in the event of a failure, compensating transactions are executed to roll back all subtransactions.

Before diving into the details of how the Saga pattern is implemented, let’s first understand the nature of the problem that Saga solves. The Saga pattern solves the problem of building distributed transactions across multiple microservices in a cloud native application.

**NOTE**

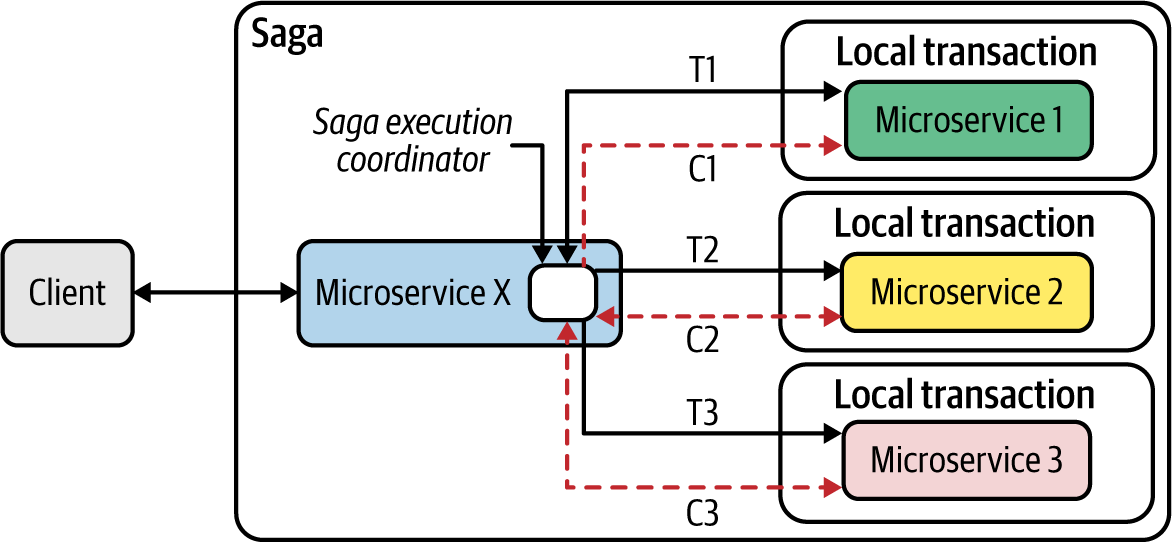
Distributed transactions use a central process called a *transaction manager* to orchestrate the steps of a transaction. The main protocol used in implementing distributed transactions is known as a *two-phase commit* (*2PC*).

Distributed transactions have inherent limitations, though, which hinder their usage with 2PC for most microservice use cases that require transactions. Some of the limitations of distributed transactions using 2PC are listed here:

* The transaction manager is the single point of failure. Pending transactions will never complete if the transaction manager is unavailable.
* If a given participant fails to respond, the entire transaction will be blocked.
* The 2PC protocol assumes that if a given participant has agreed to commit a transaction by responding Yes to the transaction manager, then it can definitely commit the transaction too. This is not the case with most of the practical scenarios. The participant may fail to commit a transaction although it has responded Yes.

Given the distributed and autonomous nature of microservices, using a distributed transaction/two-phase commit for implementing transactional business use cases is a complex, error-prone task that can hinder the scalability of the entire system.

For example, suppose we need to build a service composition scenario across multiple microservices ([Figure 3-26](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_composition_with_distributed_tr)). This composition logic must be implemented in such a way that all the invocations are done in a single transaction, which means all the transactions (T1, T2, and T3) must be executed together or not at all. The composite business capability can be built into another microservice (Microservice X) that is responsible for the execution of the distributed transaction.



**Figure 3-26. Service Composition with distributed transactions using the Saga pattern**

The Saga pattern breaks this distributed transaction scenario into multiple local transactions and groups them together to execute and introduce compensating operations to roll back each subtransaction. In [Figure 3-26](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_composition_with_distributed_tr), Microservice X is responsible for implementing the distributed transaction, and each downstream microservice can execute a local transaction (such as transactionally adding data to a database or publishing a message to a queue). To roll back these local transactions, each microservice offers a *compensating operation*.

Microservice X executes composition logic (in this case, an orchestration of multiple services) implemented through a component called the *Saga Execution Coordinator* (*SEC*)*.* This is a stateful invocation of all the required service calls (transactions T1, T2, and T3). If one of those invocations fails, the SEC logic of Microservice X can execute the corresponding compensating operations (C1, C2, and C3) to roll back everything.

Saga at the conceptual level is trivial, and most centralized workflow solutions such as Business Process Model and Notation (BPMN) solutions are built using the same terminology. But for cloud native applications, we need to implement the Saga pattern for microservices that are distributed and ephemeral (microservices can come and go, so we need to persist the transaction state). Thus, the implementation of the Saga pattern in the context of cloud native applications requires having a Saga log, which is a distributed log that the SEC component of the composite microservice interacts with.

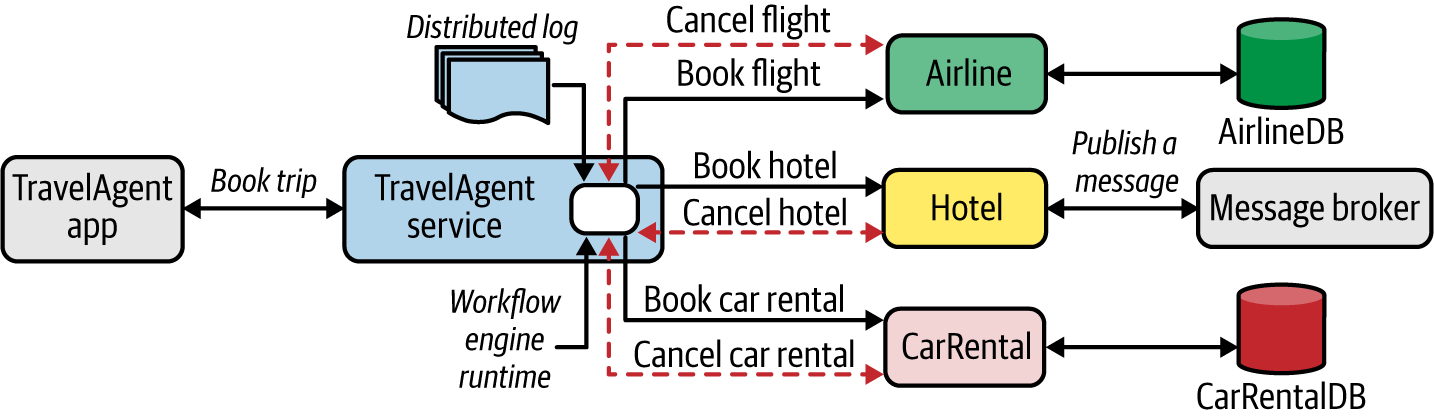
In the Saga log, we persist every transaction during execution of the given composition logic. The log contains various state-changing operations such as Begin Saga, End Saga, Abort Saga, Begin T-i, End T-i, Begin C-i, End C-i, and so on. Using these state-changing events that are persisted in a distributed log, we can roll back to any state that we want in the case of a failure. Microservice X in our scenario, in which the SEC logic is implemented, can be ephemeral because the Saga log allows us to re-create the state when booting up a new instance of the same microservice.

The SEC component orchestrates all the logic and is responsible for the execution of the Saga pattern. The SEC writes and interprets the records of a Saga log but doesn’t maintain any in-memory state. When realizing Saga patterns, we need to use a framework or technology that allows us to create the SEC inside our microservice logic, and the SEC has to connect to a distributed log that is used to maintain the state of the composition logic.

The Saga pattern is most commonly applied on top of the Service Orchestration pattern. It is possible to use the same compensating operations when using the Service Choreography pattern as well. However, in that case, there is no central SEC, but each microservice interacts with the broker transactionally. All the operations, including compensating operations, are carried out as messages/events published to a broker and events received from the broker. The transaction boundary spans only a given microservice and the broker entity (for example, a queue) that interacts with it.

**How it’s used in practice**

The Saga pattern is used when we’re building business capabilities that need distributed transactions across multiple microservices and systems. Let’s look at a real-world use case of a travel-booking service ([Figure 3-27](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#application_of_saga_pattern_in_a_travel)). Suppose we need to build a business capability that allows customers to book their airline, hotel, and car rental in a single transaction. The airline, hotel, and car-rental functionalities are implemented as microservices. Each service performs its own local transaction, such as adding an airline reservation to a database, publishing the hotel booking to a broker, and adding the car-rental reservation to a database.



**Figure 3-27. Application of Saga pattern in a travel-booking scenario**

To implement the Saga pattern, the microservices that the travel-booking service invokes should support corresponding compensating transactions such as Cancel Flight, Cancel Hotel, and Cancel Car Rental. It’s the responsibility of each microservice to ensure the safety of all the local transactions. The travel-booking service therefore uses a distributed log to record transactions and to implement the business logic. Most of the tasks related to state persistence of Saga execution and restoring after a restart are handled by the underlying Saga implementation. The business logic focuses on only the invocation of underlying business functionalities and compensating operations whenever required.

Frameworks and workflow engines such as [Camunda](https://camunda.com/) and [Apache Camel](https://oreil.ly/C1via), and cloud services such as [Azure](https://oreil.ly/V5fOR), support implementation of the Saga pattern using serverless functions and event brokers.

**Considerations**

The application of the Saga pattern for building service composition with distributed transactions should be done only when absolutely necessary. In most cases, you can avoid distributed transactions across multiple services. If your use case inherently requires distributed transactions (such as xyz), you need to be aware of these considerations when using the Saga pattern:

* The implementation of the Saga pattern requires a Saga framework or a workflow engine solution that supports stateful execution (that is, the Saga Execution Coordinator) of the business transaction between services. Implementing everything from scratch is generally not recommended because of the complexity involved.
* Running an observability solution alongside a Saga implementation is essential, as we have to debug and troubleshoot complex business transactions across distributed services.
* A Saga implementation framework should be backed by a scalable distributed log, as a single transaction may emit a multitude of events to the Saga log.

**Related patterns**

The Saga pattern is most commonly applied on top of the Service Orchestration pattern when creating service compositions. Most of the local transactions done by each service leverage data-related patterns, covered in [Chapter 4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch04.html#data_management_patterns).

**Technologies for Implementing Service Composition Patterns**

The service composition patterns are implemented using multiple types of implementation technologies. In most cloud native applications, we can create compositions by using a microservice development framework such as [Spring Boot](https://oreil.ly/0kkRx), [Quarkus](https://oreil.ly/RFg5Z), [Micronaut](https://oreil.ly/4YzhX), or [Go kit](https://oreil.ly/5fayD), or by simply using the programming language (Go, Python, Node, C#) directly. However, if the composition requires heavy lifting when it comes to integrating services with multiple protocols and messaging patterns, using a dedicated composition or integration framework such as [Apache Camel](https://camel.apache.org/) is a better option. The conventional integration platform vendors ([MuleSoft](https://oreil.ly/OSkwl), [Red Hat Fuse](https://oreil.ly/qNQxd), and [WSO2 Micro Integrator](https://oreil.ly/f6izc)) also offer cloud native variants of their platform that can be used to build service composition patterns. Such offerings are also available as *integration platform as a service* (*iPaaS*), a fully managed service for creating service compositions (for example, [Boomi](https://boomi.com/), [Azure Logic Apps](https://oreil.ly/NjbIw), and [MuleSoft Anypoint Platform](https://oreil.ly/wk6fP)). For workflows and Saga implementations, you can use dedicated workflow engines such as [Camunda](https://oreil.ly/gZV8n), [Netflix Conductor](https://oreil.ly/SrIpO), or Uber’s [Cadence](https://cadenceworkflow.io/), to execute service composition in a stateful and transactional way

**Summary of Service Composition Patterns**

[Table 3-2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#service_composition_patterns-id00202) summarizes the service composition patterns, and indicates when to use and when to not use each.

| **Pattern** | **When to use** | **When not to use** |
| --- | --- | --- |
| Service Orchestration | The business use case requires one service to handle all the interactions with other services and systems. Usually suitable for interactive services. | Not suitable if the coupling between services is a concern. Not suitable when the majority of your use case is based on asynchronous messaging for events. |
| Service Choreography | You require service composition across event-driven microservices. You want to build fully decoupled microservices in a cloud native application. | Not well suited for interactive services such as APIs that are exposed to consumers. |
| Saga | Distributed transactions across multiple microservices are essential. | Not useful when the services cannot offer compensating operations that can execute transactionally. You shouldn’t adopt it unless you have a framework or solution that can build the Saga execution for you. (Implementing Saga from scratch is overwhelmingly complex.) |
| Table 3-2. Service composition patterns | | |

**Summary**

Cloud native applications are distributed applications that consist of a collection of microservices and systems. In this chapter, we explored patterns that allow you to build connectivity among microservices as well as the other systems in your cloud native application in an efficient, resilient, secure, and scalable manner. You also learned how to use patterns to create services that require composition of multiple services and systems to realize the business capability that you want to implement. In [Chapter 7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_and_consumption_patterns), we’ll delve deeper into how these capabilities can be presented to consumers as managed APIs.