**Chapter 8. Cloud Native Patterns in Practice**

In this book, we’ve explored a lot of patterns related to building cloud native applications. Now it’s time to see how to apply some of those patterns in a real-world use case. This final chapter will show you how to apply cloud native patterns when building various aspects of cloud native applications—connecting services, managing decentralized data, event processing, stream processing, exposing managed APIs, connecting frontend applications, and performing dynamic management of the applications. Let’s begin by looking at the details of the use case that we are trying to implement.

**Building an Online Retail System**

As our real-world use case, we chose a simple online retail application that we can use to search for goods, purchase them, and ship them to our preferred locations. While an actual retail application can be overwhelmingly complex, we have selected a simplified yet diverse set of business capabilities and requirements to better demonstrate the application of cloud native patterns. We can categorize the key requirements and capabilities of this system as follows: product catalog, order management, order tracking and prediction, product recommendations, and customer and partner management. Let’s look at each of these business capabilities and requirements in more detail.

**Product Catalog**

We need to build a product catalog that allows customers of the online retail application to search for products and obtain essential product details to make a purchase decision.

The key functionality of the product catalog includes the following:

* Search for products.
* Obtain the product details to make a purchase decision.
* Enable product administrators and sellers to add, remove, and update products.

**Order Management**

Once the customer decides to purchase a given product, they select it and the required quantity and add it to the shopping cart. Then the customer should be able to place an order for those items. The key functionality of the order management subsystem includes the following:

*Place an order*

After a customer decides to purchase a given product with the required quantity, they should be able to place an order.

*Payment*

When placing an order, the customer should be able to pay for that order by using a payment method of their choice.

**Order Tracking and Prediction**

Once an order is placed, the customer should be able to obtain or receive order and shipment updates and notifications. They key functionalities of the order tracking and prediction subsystem include the following:

* Track the status of the shipment.
* Get notification on the changes to order status, delivery estimates, and other important changes.
* Provide a delivery prediction: the system should be able to predict the delivery time based on the past order, real-time data from warehouses, and shipping services.

**Product Recommendations**

A customer should get recommendations of various products based on their search queries, past purchases, and currently available deals.

**Customer and Partner Management**

A customer should be able to manage their accounts and profile information. The customer may be using various types of applications (such as mobile or web) to access the system. Certain business capabilities may be exposed to multiple parties such as customers and partners via different frontend applications (customer mobile app, partner mobile app, and so on).

Also, as nonfunctional requirements, we can assume that the entire application needs to be highly available, scalable, secure, observable, and dynamically manageable across an on-premises data center and a public cloud platform.

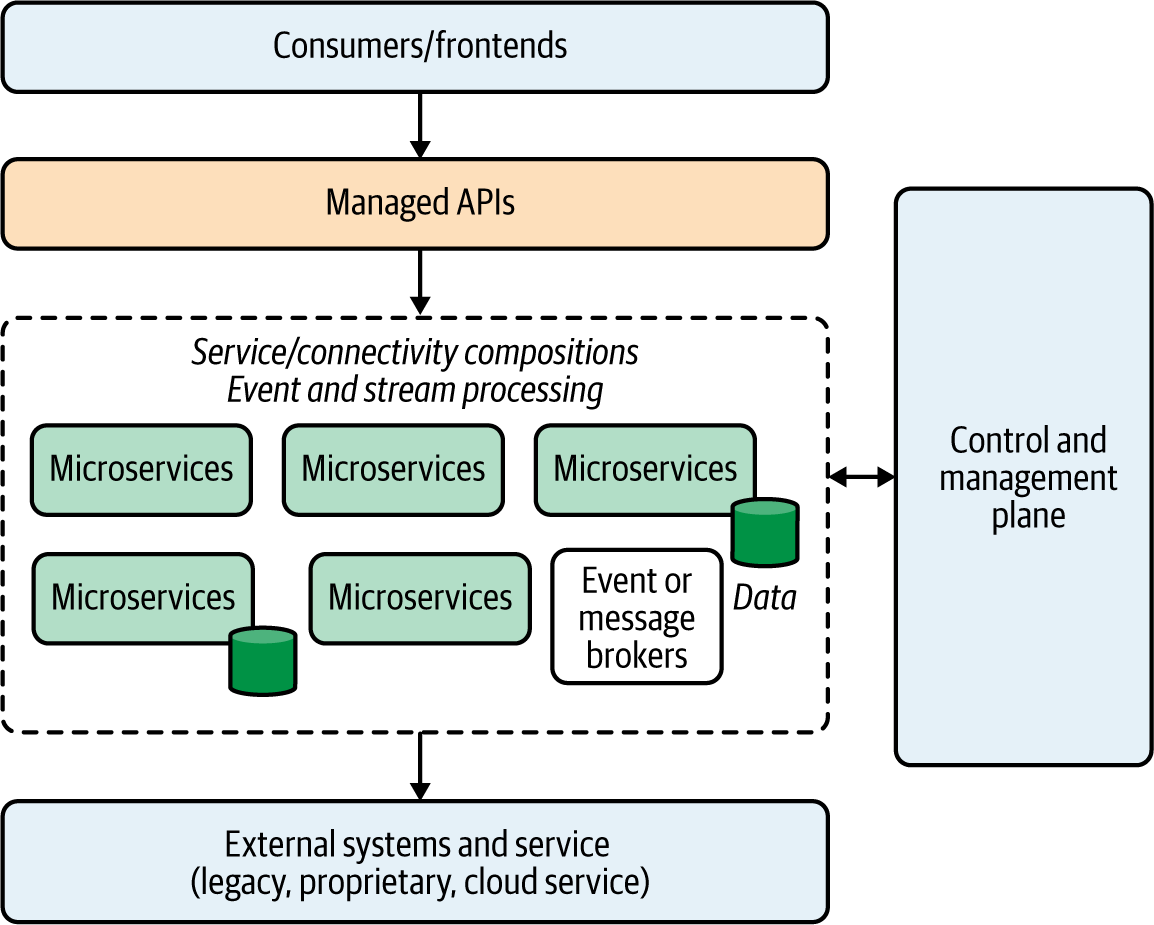
Based on these business requirements and capabilities, we can build the solution as a cloud native application by applying certain patterns that we explored in previous chapters. Let’s start by building the high-level architecture of the system.

**Building the High-Level Architecture**

The high-level architecture of the online retail application is shown in [Figure 8-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch08.html#high_level_architecture_of_the_online_r). As we have clearly identified the requirements and use cases, we can map those into the business capabilities (such as product search, order placing, payments, order tracking, and so forth) that the application needs to expose to its consumers. As you learned in [Chapter 7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_and_consumption_patterns), these capabilities can be consumed by frontend applications directly or through managed APIs. To give more controlled and managed access to the capabilities, we will expose them as managed APIs.

The architecture comprises the frontend layer, then the API management layer that connects with the rest of the system. All web applications and mobile applications of the retail system are built on top of that API layer. These APIs are backed by the microservices that implement the business logic for each capability.

The interservice communication among these services may be implemented using communication patterns such as Service Connectivity, composition patterns, or event-driven patterns. These microservices can connect to external and internal systems (e.g., a legacy ERP application) that are not implemented as cloud native applications. The entire system should support cross-cutting concerns such as scalability, high availability, security, and observability through a control plane.



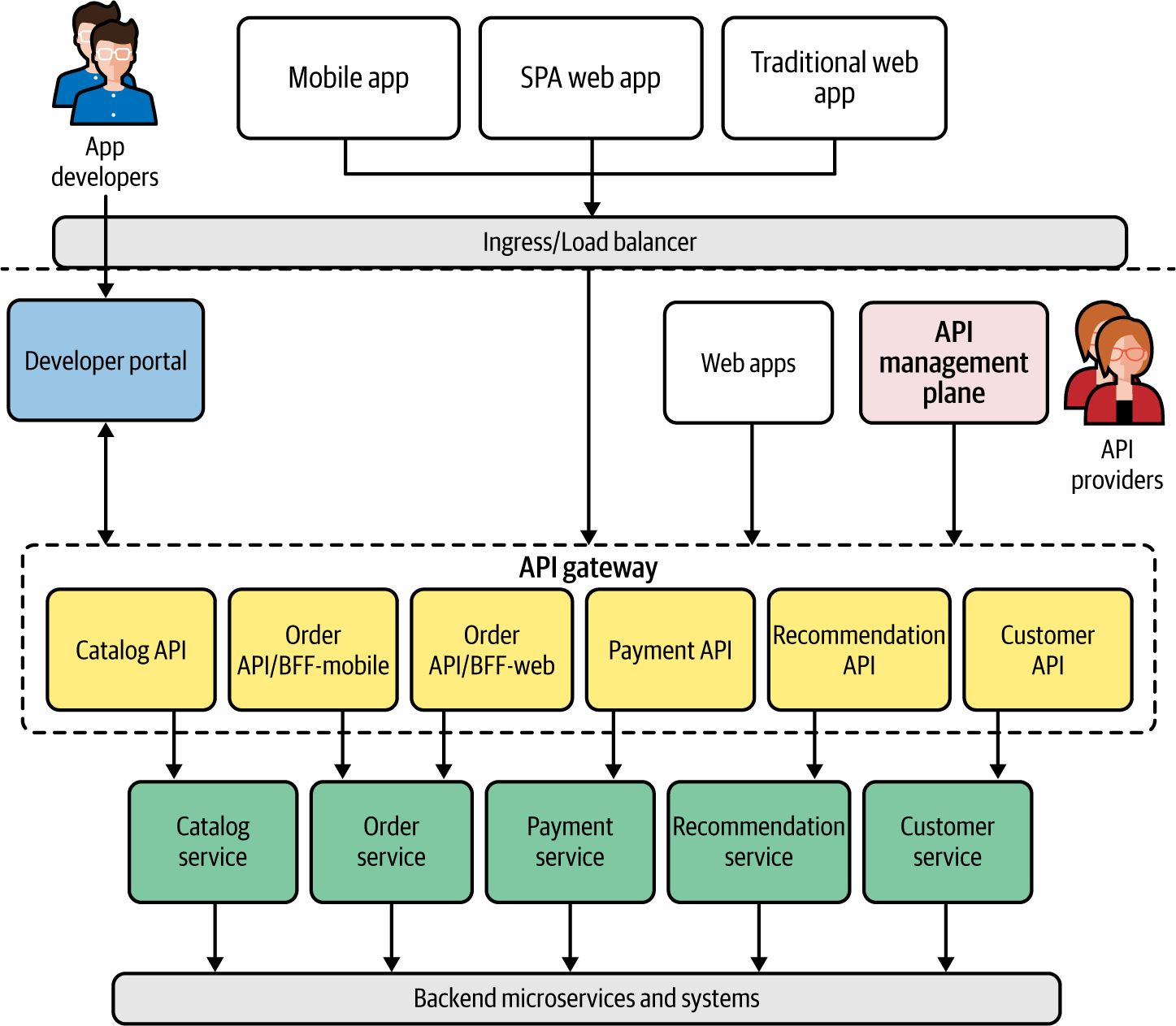
**Figure 8-1. High-level architecture of the online retail cloud native application**

Now, let’s dive into each aspect of this cloud native application and discuss the details of the implementation.

**Building External APIs**

Since our focus is on realizing a set of business capabilities, it makes sense to start from the business capabilities that we want to expose to the consumer of the retail application. These business capabilities are essentially the APIs that we would expose from our retail application. These APIs can be used by the frontend application of the organization or by any external applications such as partner applications. Therefore, let’s start building this cloud native application by looking at how to model the frontend applications around those APIs (we are following the top-down approach of API management covered in [Chapter 7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_and_consumption_patterns)).

We identify the key business capabilities that we want to expose: product catalog, order management, payments, recommendations, tracking, and delivery prediction ([Figure 8-2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch08.html#designing_apis_and_frontend_application)). We then model these capabilities as managed APIs and expose them to the external parties, such as consumers and partners, via an API management plane. The microservices that back those APIs implement the required business logic by integrating with other microservices, external systems, databases, and other resources. In this use case, we apply API management patterns such as API Gateway (discussed in [Chapter 7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_and_consumption_patterns)) to expose these services as managed APIs.



**Figure 8-2. Designing APIs and frontend applications**

These APIs are accessible to external client applications, such as mobile and web applications, through a load-balancer layer. The frontend application development can leverage patterns such as Backend for Frontends to implement certain features. For example, the order API may have different BFF-style APIs for mobile and web applications. As we discussed in [Chapter 7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_and_consumption_patterns), most of the API traffic can be exposed as RESTful or GraphQL services through the API gateway. The cross-cutting capabilities such as security, throttling, caching, and policy enforcement are applied at the API gateway layer. The API runtimes that we implement at the gateway layer may use patterns such as API Microgateway to enable independent runtimes for each API.

The ownership of these APIs can be assigned to the team that owns the underlying microservice backing a given API. For example, if the Catalog service is owned by Team A, the same team can own the catalog API as well. The key idea here is to segregate the API management layer based on ownership so that we can change, scale, and manage APIs independently. Therefore, you may opt to use patterns such as API Microgateway to decentralize the gateway layer further.

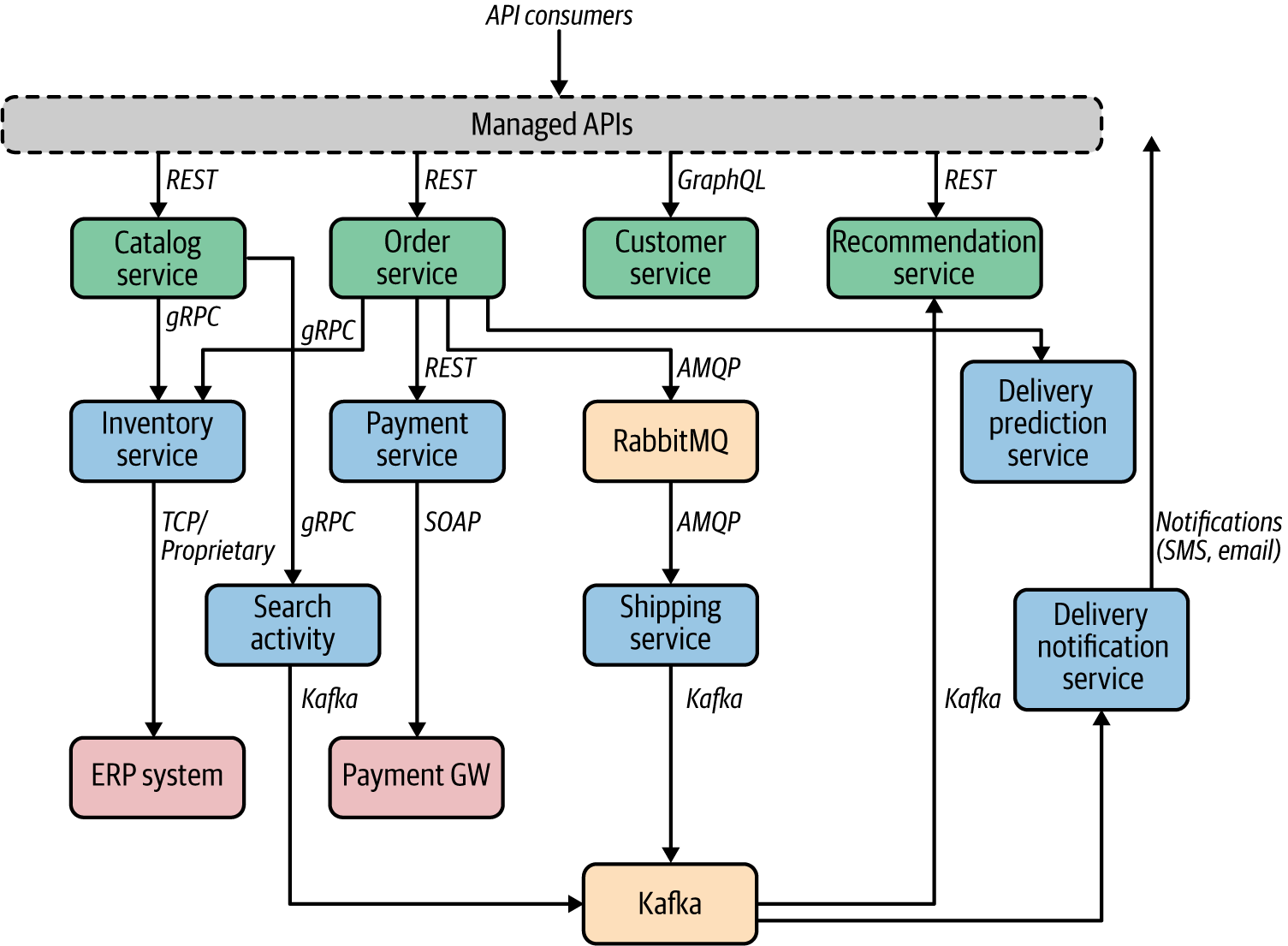
Once we decide on the APIs that we need to expose at the API management plane, we can dive into the implementation of each of the microservices that are backing those APIs.

**Connecting Services**

In order for our microservices to implement the business logic (such as order management, payments, and customer management), we need to integrate and connect multiple microservices and systems. Based on the APIs that we’ve identified, we can design the downstream microservices such as Order service, Catalog service, and Customer service.

These services invoke other services, such as Inventory and Payment, while also communicating with systems such as ERPs, message brokers, and databases. Therefore, we should pick the most suitable communication protocol to implement a given business use case and use it for service development.

For example, as you can see in [Figure 8-3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch08.html#connecting_services_and_systems_in_our), the Order service exposes a RESTful API, while the Customer service uses GraphQL. Also, the Order service accepts messages through a synchronous protocol such as REST/HTTP, while it asynchronously enqueues the order request with AMQP using RabbitMQ.



**Figure 8-3. Connecting services and systems in our online retail application**

The internal services can communicate by using high-performance synchronous messaging techniques such as gRPC.

Most of the connectivity patterns that you learned in [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern) need to be applied when connecting these services. For example, resilient connectivity techniques such as time-outs, retries, and circuit breakers need to be applied when invoking services. If the deployment runs on top of a service mesh, most of the interservice communication can be managed and observed via the service mesh control plane. When it comes to creating compositions, we have used a hybrid of the Service Orchestration and Service Choreography patterns. For example, the Order service is orchestrating the composition across multiple other services, while the Shipping service operates in a reactive way (based on messages or events).

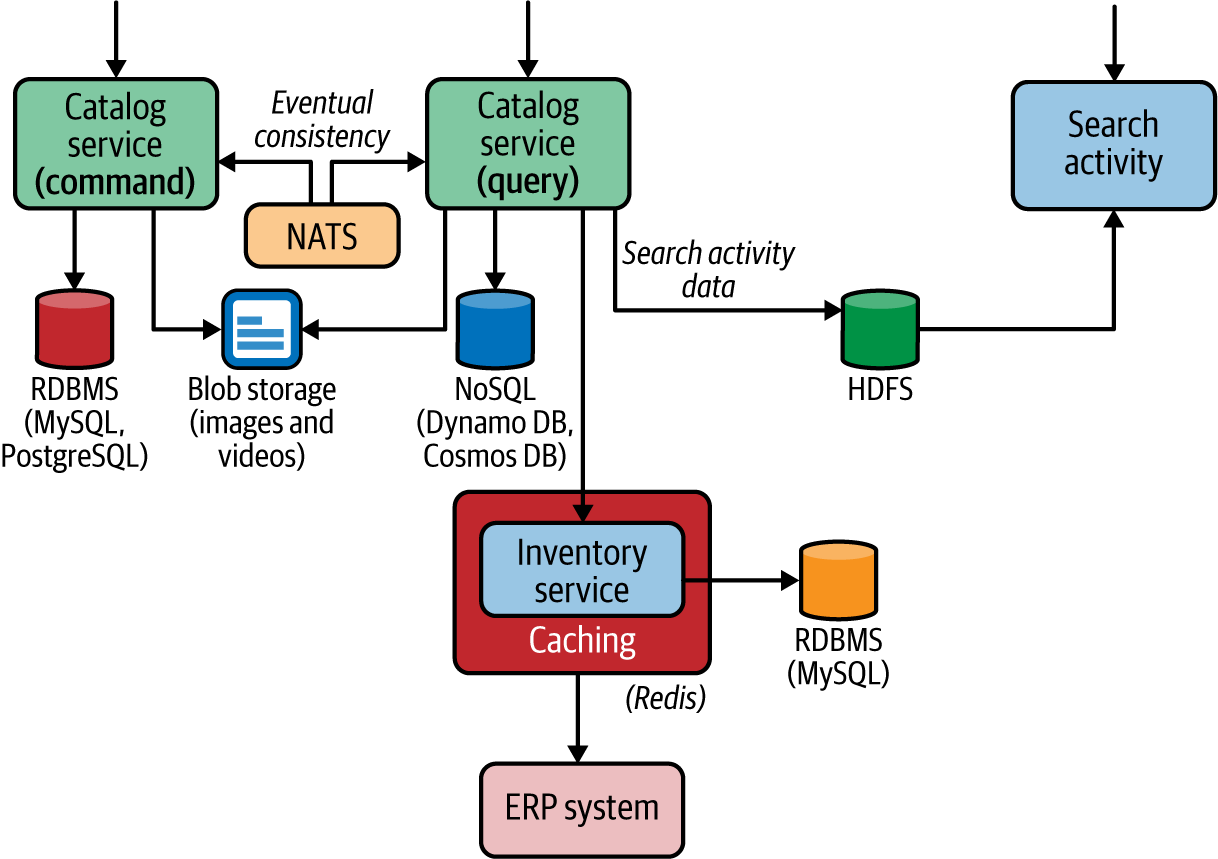
Now, let’s dive into some of the details of data management in this use case.

**Performing Data Management**

As we explored in [Chapter 4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch04.html#data_management_patterns), one of the key aspects of cloud native data management is to manage data decentrally at the microservice level. This approach removes most of the coupling between microservices and gives the microservice owners the freedom to choose the most suitable data management technique. In our retail application, we can use these principles to manage data to implement the requirements that we discussed earlier. Rather than focusing on how data management happens across the entire application, let’s look at specific use cases of data management.

Let’s focus on how data management is done at the Catalog and Inventory services ([Figure 8-4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch08.html#decentralized_data_management_with_priv)). The Inventory service has a dedicated relational database for building the inventory capabilities, while some services such as Catalog that do high-intensive data writes use a NoSQL database. We can apply patterns such as CQRS to split the capabilities of the Catalog service into multiple independent services, which allows us to have command and query components. The command part can be implemented using an RDBMS, while the query part leverages a NoSQL database underneath.

The eventual consistent data synchronization between these two service components is achieved via *eventing*, in which we use an event broker such as NATS or Kafka. With this approach, the command and query parts of the Catalog service are completely decoupled, and we can scale them independently based on our business needs (for instance, the Catalog query service may get much higher load compared to the Catalog command service).



**Figure 8-4. Decentralized data management with private databases, CQRS, caching, and heterogeneous databases**

Based on the type of data that we have to store, we may opt to use dedicated data stores. For example, suppose we have to store images and videos of the products; then we can use a dedicated and optimized binary data store such as a blob store and include references from the product database. We can also apply patterns related to caching for the inventory service, where we use a caching layer between the inventory and the legacy ERP system.

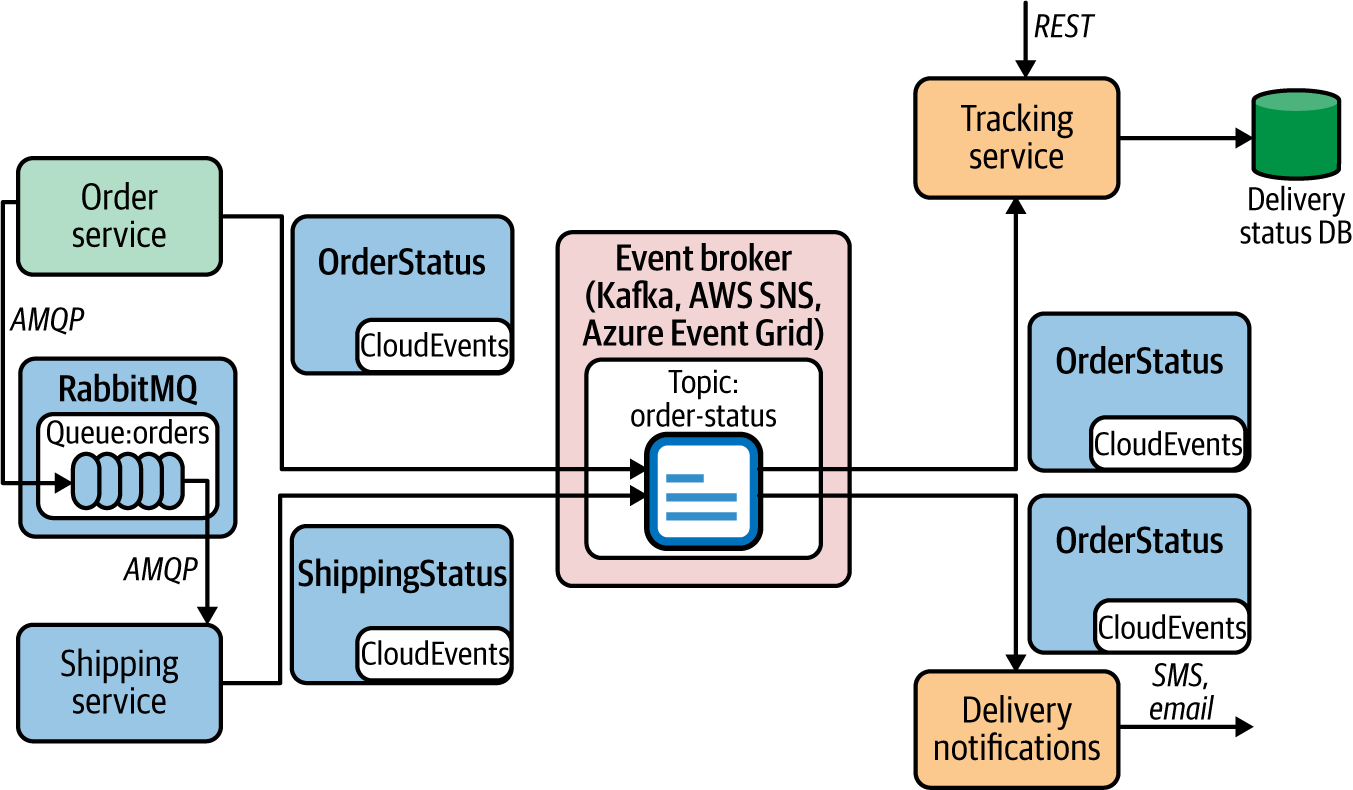
**Using Event-Driven Architecture**

Order status management is one of the key business capabilities; we need to keep track of all state-changing events related to orders and allow the consumers (such as Shipping and Tracking services) to process those events independently. This requires the application of event-driven patterns that we explored in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns). Let’s look at how we can use those patterns when realizing order status management across online retail applications.

The Order service uses an event-driven approach to propagate the order status across multiple microservices ([Figure 8-5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch08.html#event_driven_communication_with_event_s)). A given order can have multiple states that are managed by various microservices during the order life cycle. When an order is created, the Order service enqueues the order message, which creates a *command* to the consumers of that queue to process the order request.

The Shipping service consumes the message from the *orders* queue. When the Order service enqueues the message, the order status update event is published to a distributed event log via an event broker. So, we use the Event Sourcing pattern to record all the life-cycle state-changing events of the order processing. This distributed event log can reside on an event broker such as Kafka, Amazon SNS, or Azure Event Grid.

Because the events related to orders come from multiple sources, we can use a common event representation convention such as CloudEvents. With CloudEvents-based events, we use common metadata across all the event types that we use for event sourcing.



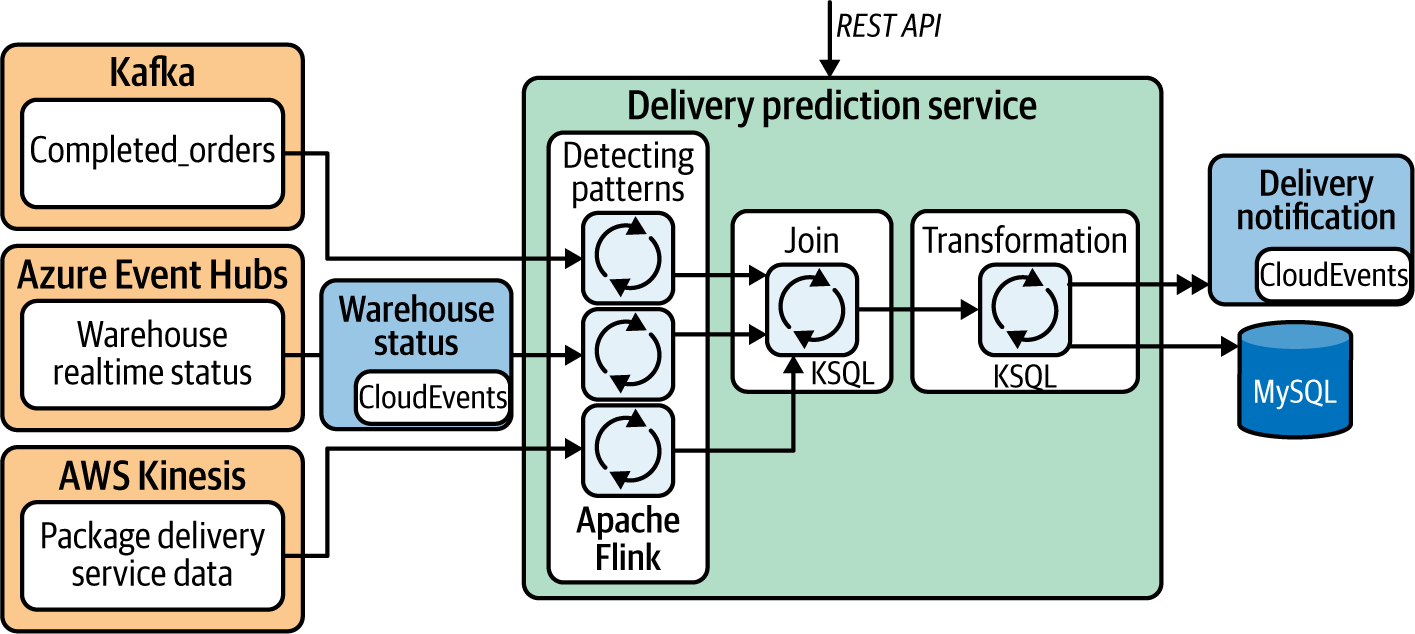
**Figure 8-5. Event-driven communication with event sourcing and CloudEvents**

The subscribers of the order status events can consume the event notification through the event broker component. Because we use CloudEvents, the original event’s metadata is preserved in a standard format that can be used across multiple services and systems. For example, the Tracking service can consume those events and store the required information in a data store to serve the consumers of the Tracking service, while the Delivery Notifications service uses the event log to notify customers of order and shipment changes via text messages or email. Since we record all order-related events in a distributed log, if there are future business requirements related to order processing, we can simply extend the support for those use cases by adding a new event consumer at the event-broker level that can independently process the event log.

**Using Stream Processing**

The delivery-prediction capability of our online retail application can use the stream-processing patterns from [Chapter 6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#stream_processing_patterns-id00204). The main business capability here is to build a service that predicts the delivery time of an order based on multiple input data sources such as real-time warehouse data, data from past orders, and real-time information from shipping companies.

We can design a microservice that can process events originating from multiple sources such as IoT devices or sensors. These events can then be ingested by a distributed event broker or data store. Our Delivery Prediction microservice uses a disparate set of event stream sources (Kafka, Azure Event Hubs, Amazon Kinesis), as shown in [Figure 8-6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch08.html#processing_multiple_streams_of_data_and).



**Figure 8-6. Processing multiple streams of data and predicting the delivery time**

The business logic of the service consists of identifying delivery patterns by processing a continuous stream of delivery-related events in real time (using patterns such as Filters and Thresholds and Windowed Aggregation), and then joining (via the Stream Join pattern) and transforming (via the Stream Transformation pattern) them before we send the notification to the event-sink service or systems.

The stream-processing logic can be built using multiple stream-processing technologies based on your use case. Here we have used Apache Flink and KSQL. The results can also be stored in a data store such as an RDBMS to serve the consumers of the Delivery Prediction service. Therefore, this service can serve any request regarding a specific delivery as well as send a notification to interested parties on updates to delivery times, and so on. (This is a place where you can apply artificial intelligence and machine learning patterns to predict delivery time as well. However, we don’t go into the details of these patterns as they’re beyond the scope of this book.)

**Implementing Dynamic Management in a Cloud Environment**

So far, we have discussed the implementation of various business capabilities and requirements for our online retail application using cloud native design patterns. However, as we discussed in [Chapter 1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#introduction_to_cloud_native), we also need to deploy, run, manage, and govern our application. Often this can be achieved using existing cloud services. You could also build these capabilities from scratch with your own data center.

Let’s suppose we are using a cloud provider such as AWS, Azure, or GCP to build this application. We can use the cloud platform to implement the following nonfunctional requirements of our online retail application:

*Technology*

The microservices of the online retail application can be developed using any technology of your choice. For instance, some of these microservices can be implemented using a technology stack such as Spring Boot or .NET, or you can use serverless functions offered from the cloud service.

*Automation of development, release, and deployment*

We can simply use the provided development and CI/CD service of the cloud platform to achieve this.

*Run on a dynamic runtime environment*

Since the retail application needs to scale and be highly available, we can use the container and container orchestration service (e.g., Kubernetes service) of the cloud provider.

*Use supporting backing services*

We need quite a few backing services such as event brokers, databases, and caching components to realize our business capabilities. When you are using a cloud platform, you get all these services as managed services with all the nonfunctional capabilities including high availability, security, autoscaling, and resilience.

*Service connectivity*

When you deploy and run a service on a cloud platform, you often get the service-connection-related features such as service discovery, service resiliency, routing, and even a full-fledged managed service mesh as a cloud service.

*Security*

The application that you deploy on such a cloud platform can be secured using transport-level security or application-level security protocols such as OAuth2 or OpenID Connect.

*Observability*

When the application is running, you should be able to get metrics, logs, and other operation data through the observability layer of the cloud service.

*Single control plane for hybrid and multicloud deployments*

It is not mandatory to stick to a single cloud platform. If you have workloads in multiple clouds or even in your own data center, you can use a control from one cloud platform to manage the other workloads. For example, you can use offerings such as Azure Arc or Google Anthos for this purpose.

**Summary**

In this chapter, we applied certain patterns presented earlier in the book to build the key components of an online cloud native retail application. Our primarily focus was on exposing business capabilities as managed APIs, establishing service connectivity and creating compositions, building event-driven capabilities using EDA, using stream processing to handle specific data-stream-processing capabilities, and finally running the entire application on top of a dynamically managed cloud native application using a cloud platform.

While the focus of the book has been on *designing and building* cloud native applications, we encourage you to seek out additional resources to gain an understanding of other aspects of cloud native applications, such as security, deployment, and DevOps.