**Chapter 1. Introduction to Cloud Native**

The software development landscape is constantly changing and evolving through modern architectural paradigms and technologies. From time to time, software architecture goes through a fundamental shift with the emergence of breakthrough technologies and approaches. One such breakthrough is *cloud native architecture*. It is such a major shift in the context of software application development, one that changes the way we build, ship, and manage software applications. Cloud native architecture has become an enabler of agility, speed, safety, and adaptability for software applications.

This chapter helps you understand what cloud native is by exploring the key characteristics of cloud native applications. We’ll also introduce a development methodology that you can use throughout the life cycle of cloud native applications. Then we’ll focus on the importance of using design patterns for developing cloud native applications. Let’s begin our discussion by defining *cloud native*.

**What Is Cloud Native?**

So, what’s the formal definition of *cloud native*? The sad news is, there’s no such definition. Cloud native means different things to different people. The closest general definition is from the Cloud Native Computing Foundation (CNCF), an organization dedicated to building sustainable ecosystems and fostering communities to support the growth and health of open source, cloud native applications. CNCF serves as the vendor-neutral home for many of the fastest-growing open source projects that can be used in building cloud native applications.

**CLOUD NATIVE DEFINITION FROM CNCF**

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach. These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

For the purposes of this book, we take a bottom-up approach to defining *cloud native*. We look at all the characteristics of cloud native applications, across the board, by going through every stage in the life cycle of a cloud native application—including design, development, packaging, deployment, and governance. Based on those characteristics, we’ve come up with the following definition:

*Cloud native is building software applications as a collection of independent, loosely coupled, business-capability-oriented services (microservices) that can run on dynamic environments (public, private, hybrid, multicloud) in an automated, scalable, resilient, manageable, and observable way.*

Exploring these characteristics further helps us understand cloud native applications. Let’s look more closely at the characteristics in our definition.

**Designed as a Collection of Microservices**

A cloud native application is designed as a collection of loosely coupled and independent services that are serving a well-defined business capability. These are known as *microservices*. Microservices are the foundational architectural principle that is essential to building cloud native applications. It’s virtually impossible to build a proper cloud native application without knowing the basics of microservices architecture.

*Microservices architecture* is a style of building software applications. Before the advent of microservices architecture, we used to build software applications as monolithic applications catering to various complex business scenarios. These monolithic applications are inherently complex, hard to scale, expensive to maintain, and hinder the agility of development teams. Monolithic applications communicate with one another by using proprietary communication protocols and often share a single database.

**TIP**

Microservices architecture is about building a software application as a collection of independent, autonomous (developed, deployed, and scaled independently), business-capability-oriented and loosely coupled services.[**1**](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#idm46399015392472)

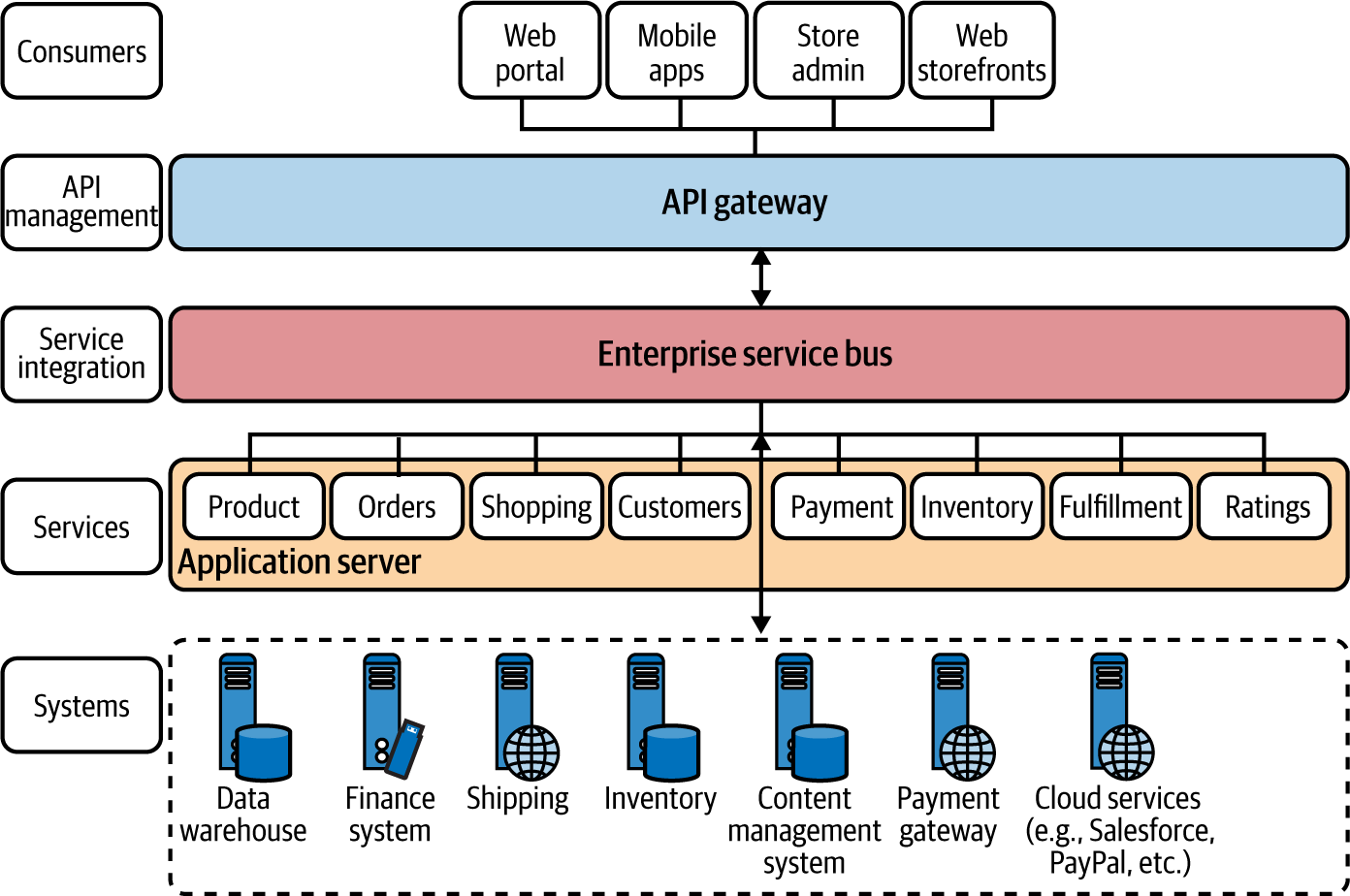
*Service-oriented architecture* (*SOA*) emerged as a better architectural style to address the limitations of the monolithic application architecture. SOA is built around the concept of modularity and building a software application as a collection of services to serve a specific business capability. The realizations of SOA, such as web services, were implemented using complex standards and message formats, and introduced centralized monolithic components into the architecture.

In a typical SOA-based design, software applications are built using a set of coarse-grained services, such as web services, that often leverage open standards and a central monolithic integration layer known as the *enterprise service bus* (*ESB*). An API management layer can be used on top of this architecture so you can expose the capabilities as managed APIs.

[Figure 1-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#an_online_retail_application_scenario_b) shows a simple online retail application designed using SOA. All the business capabilities are created at the services layer as coarse-grained services that run on a monolithic application server runtime. Those services and the rest of the systems are integrated using an ESB. Then an API gateway is placed as the front door to the SOA implementation, where you control and manage your business capabilities.

This approach has worked for many enterprises, and a lot of enterprise software applications are still built using SOA. However, its inherent complexities and limitations hinder the agility of developing software applications. Most implementations of SOA result in a lack of independently scalable applications, inter-application dependencies that hinder independent application development and deployment, reliability issues of being a centralized application, and constraints on using diverse technologies for the application.

Microservices architecture, on the other hand, eliminates the limitations of SOA implementations by introducing more fine-grained and business-oriented services while eliminating centralized components such as ESB. In microservices architecture, a software application is designed as a collection of autonomous and business-capability-oriented services that are developed, deployed, and often managed independently by different teams. The granularity of the service is determined by the application of concepts such as the bounded context in the Driven Design paradigm.[**2**](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#idm46399015379896)

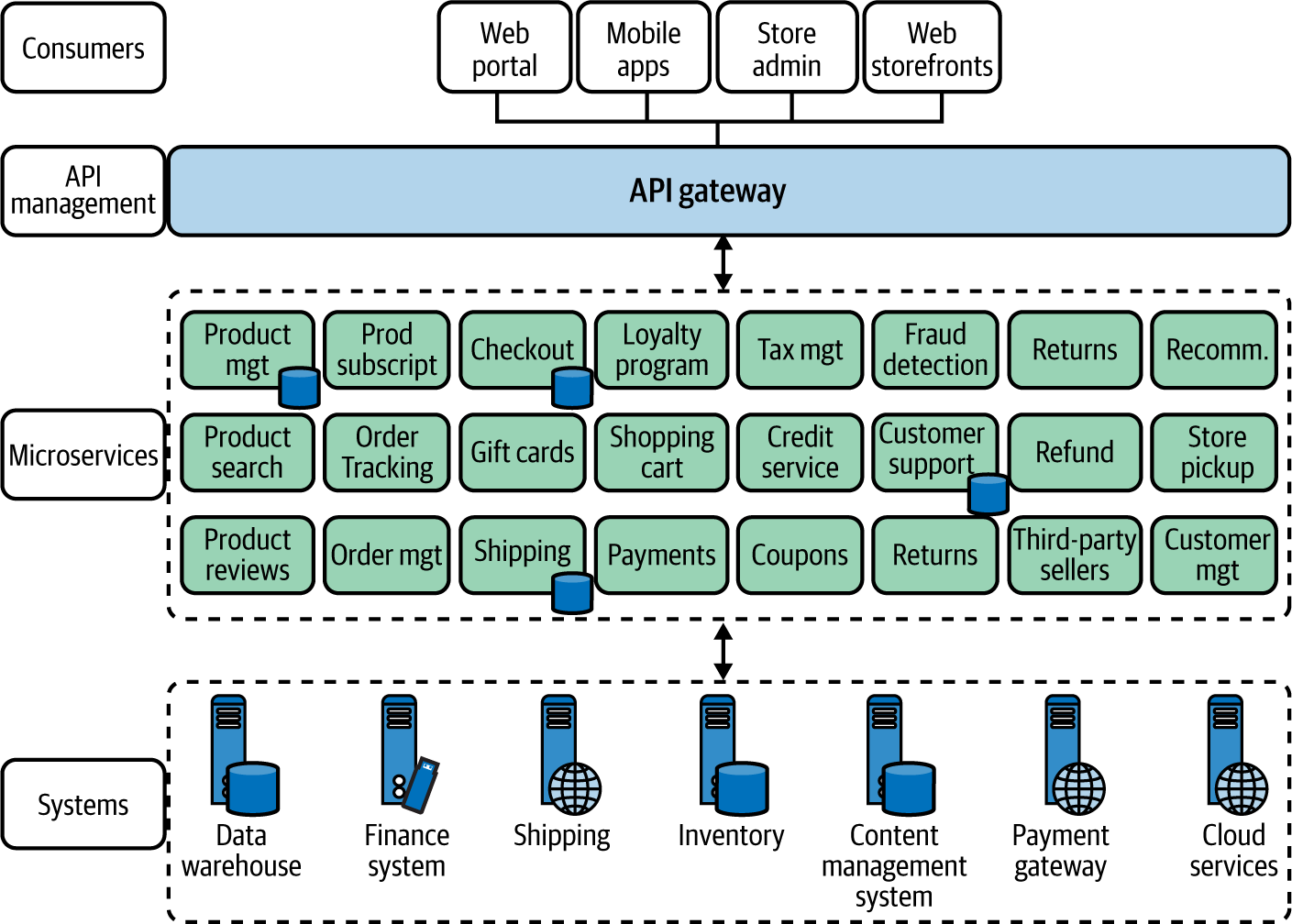


**Figure 1-1. An online retail application scenario built using an SOA/ESB with API management**

We can transform our earlier SOA/ESB-based online retail application to microservices, as shown in [Figure 1-2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#an_online_retail_application_built_usin). The main idea here is to introduce microservices for each business capability that we identify during the design phase, as we apply the concepts of domain-driven design (explained later in this chapter) and eliminate the centralized integration at the ESB layer.

**NOTE**

Monolithic-to-microservice transformation techniques are discussed in detail in [*Building Microservices*](https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/) by Sam Newman (O’Reilly).



**Figure 1-2. An online retail application built using microservices architecture**

Rather than using an ESB layer to integrate the services, microservices themselves create the compositions through lightweight interservice communication that’s required to build the business capability offered by the microservice. Therefore, these microservices are called *smart endpoints* that are connected via *dumb pipes*, which refers to the lightweight interservice communication techniques.[**3**](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#idm46399015350344) Microservices may connect to other existing systems and in some cases may expose a simplified interface (often known as a *facade*) for those systems as well.

The microservices don’t share databases, and external parties can access the data only via the service interface. Each microservice needs to implement the business logic as well as the interservice communication features that include resiliency, security, and so on.

As cloud native applications are designed as a collection of microservices, almost every concept that you apply in microservices relates to the cloud native context as well. Therefore, we discuss most of the patterns and fundamentals of microservices architecture throughout the book.

**Use Containerization and Container Orchestration**

Just as microservices are important in the phase of designing and developing cloud native applications, containers are important in the packaging and running of cloud native applications. When developing cloud native applications, the microservices that we build are packaged into container images and executed on top of a container host. Let’s dig deeper to understand what this really means.

**What are containers?**

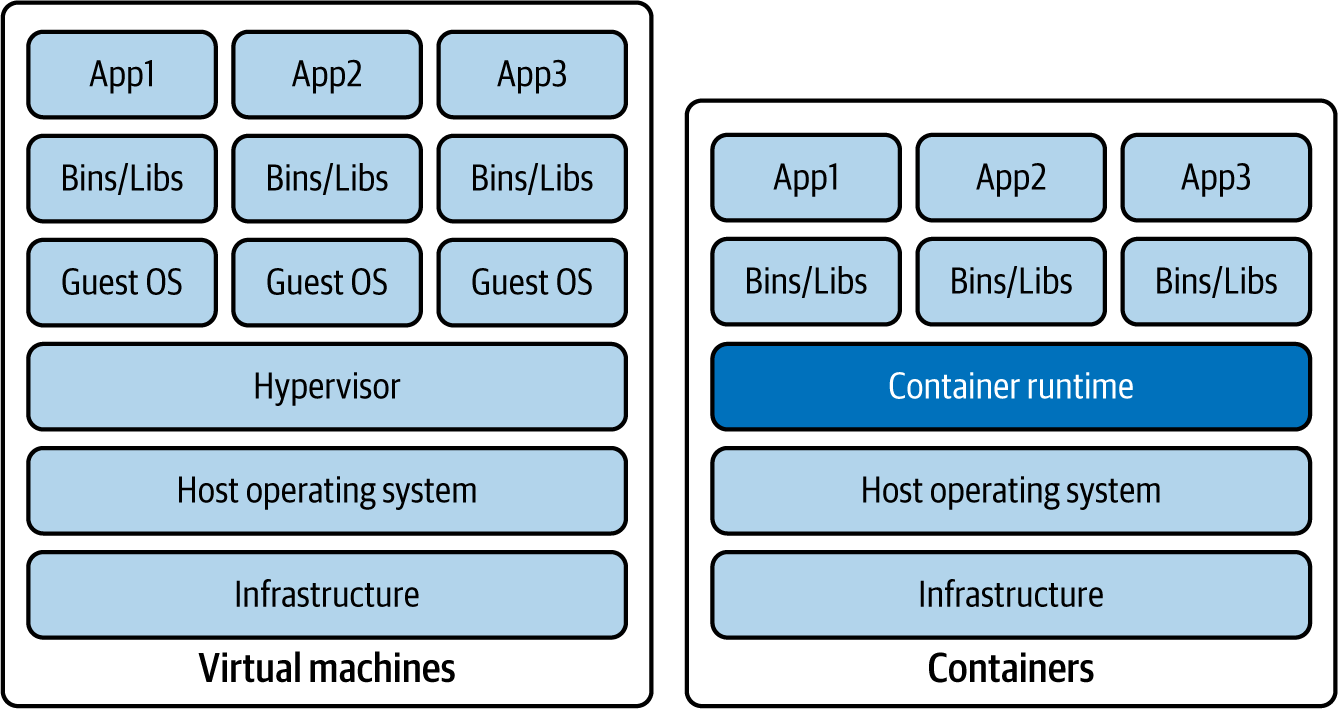
A *container* is a running process that is isolated from the host operating system and other processes in the system. A container interacts with its own private filesystem, which is provided by a container image. The container image is a binary that is formed by packaging everything that’s needed to run an application: application code, its dependencies, and runtime. These container images are immutable and often stored in a repository known as a *container registry*.

To execute a container, you can create a running process out of the container image, which is known as a *container instance*. The container instance runs on top of the container runtime engine.

[Figure 1-3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#comparing_application_execution_on_virt) compares the execution of three microservice runtimes on virtual machines (VMs) versus on a container runtime engine. Running microservices as containers is drastically different from the conventional VM execution that runs a full-blown guest operating system with virtual access to host resources through a component known as a *hypervisor*. Since containers run on top of a container runtime, they share the kernel of the host machine, processor, and memory with other containers. Hence, running microservices on containers is a lightweight, discrete process compared to running them on top of a VM. For example, an application that runs on a VM and takes several minutes to load may take only a few seconds to load in containers.

The process of converting microservices or applications to run on top of containers is known as *containerization*. Docker has become the de facto platform for building, running, and sharing containerized applications.

Containerization makes your microservices portable and guarantees execution consistency across multiple environments. Containers are a key driving force to make microservices independent and autonomous as they are self-sufficient and encapsulated, allowing you to replace or upgrade one without disrupting others, while utilizing the resources better than VMs. They also eliminate additional runtime preconfiguration, and are much more lightweight compared to VMs.



**Figure 1-3. Comparing application execution on virtual machines versus containers**

Containerization of your microservices and running them by leveraging a container engine is only one part in the development life cycle of your cloud native application. But how do you manage your containers’ execution and the life cycle of the containers? That’s where container orchestration comes into the picture.

**Why container orchestration?**

*Container orchestration* is the process of managing the containers’ life cycle. When you operate real-world cloud native applications, it’s nearly impossible to manually manage containers. Hence, a container orchestration system is an essential part of building a cloud native architecture.

Let’s have a close look at some of the key features and capabilities of a container orchestration system:

*Automatic provisioning*

Automatically provisions container instances and deployment of containers

*High availability*

Automatically reprovisions containers when one container runtime fails

*Scaling*

Based on the demand, automatically adds or removes container instances to scale up or scale down the application

*Resource management*

Allocates resources among the containers

*Service interfaces and load balancing*

Exposes containers to external systems and manages the load coming into the containers

*Networking infrastructure abstractions*

Provides a networking overlay to build communication among containers

*Service discovery*

Offers built-in capability of discovering services with a service name

*Control plane*

Provides a single place to manage and monitor a containerized system

*Affinity*

Provisions containers nearby or far apart from each other, helping availability and performance

*Health monitoring*

Automatically detects failures and provides self-healing

*Rolling upgrades*

Coordinates incremental upgrades with zero downtime

*Componentization and isolation*

Introduces logical separation between various application domains by using concepts such as namespaces

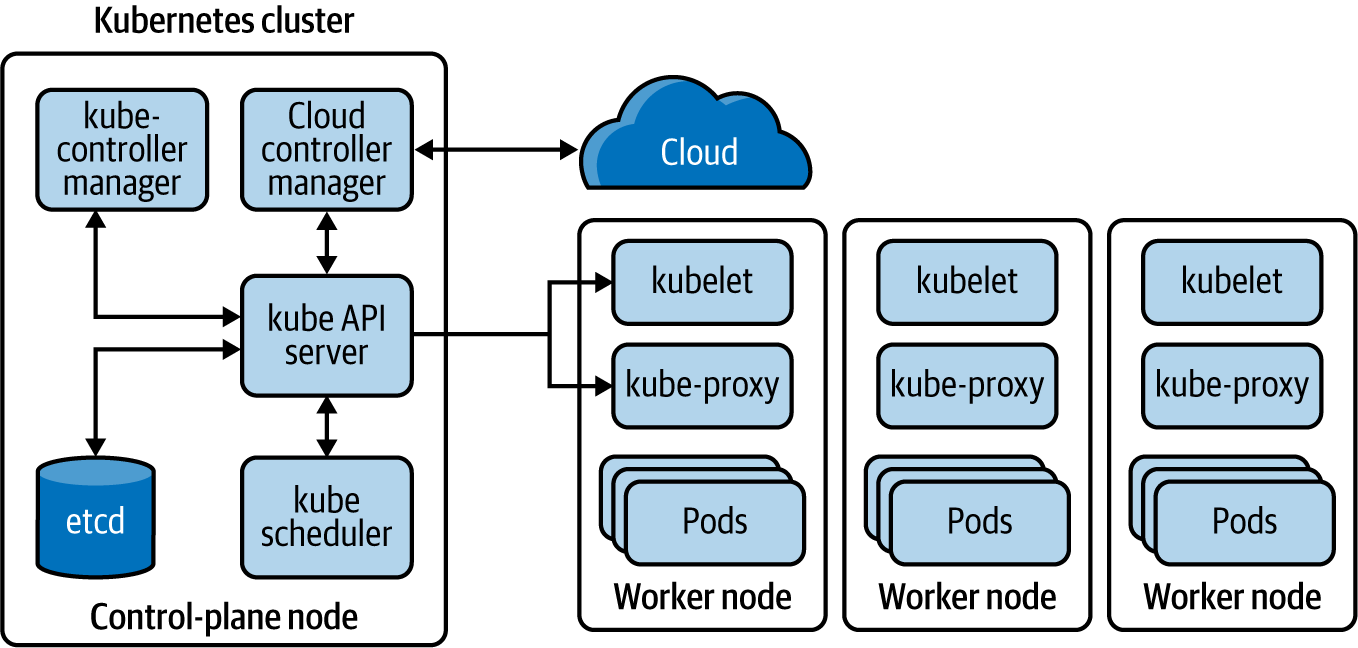
In the cloud native landscape, Kubernetes has become the de facto container orchestration system.

**Kubernetes**

*Kubernetes* creates an abstraction layer on top of containers to simplify container orchestration by automating the deployment, scaling, fault tolerance, networking, and various other container management requirements that we discussed earlier.

Since Kubernetes is adopted across multiple platforms and cloud vendors, it’s becoming the universal container management platform. All the major cloud providers offer Kubernetes as a managed service.

Applications designed to run on Kubernetes can be deployed on any cloud service or on-premises data center that supports Kubernetes, without making any changes to the application (as long as you don’t use any platform-specific features such as load balancers). Kubernetes makes application workloads portable, easier to scale, and easier to extend. It is now the standardized platform that you can design your application to, so that it won’t be coupled to any underlying infrastructure. Kubernetes brings in key abstractions that help standardize applications and simplify container orchestration ([Figure 1-4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#fundamental_components_of_a_kubernetes)).



**Figure 1-4. Fundamental components of a Kubernetes platform**

A Kubernetes cluster comprises a set of nodes that run on virtual or physical machines. Among these nodes is at least one control-plane node and several worker nodes. The *control-plane node* is responsible for managing and scheduling application instances across the cluster. Therefore, the services that the Kubernetes control-plane node runs are known as the *Kubernetes control plane*.

The Kubernetes API server takes care of all the communication between the control-plane and worker nodes. When a certain workload needs to be assigned to a given node, the *kube-scheduler* assigns workloads to each worker node based on the available resources and policies. Each Kubernetes node runs an agent process known as a *kubelet*, which maintains the node states. This is the component that directly communicates with the Kubernetes API server, receiving instructions as well as reporting states of each node.

A *pod* is the basic deployment unit representing an application runtime that runs on a given node. One pod can have one or more containers running inside it. A pod is assigned a unique IP address within the Kubernetes cluster.

Kubernetes further simplifies application deployment and management by introducing abstractions such as Service, Deployment, and ReplicaSet. A *Service* provides a logical grouping for a set of pods as a network service, so that one service can have multiple load-balanced pods. A *ReplicaSet* defines the number of replicas the application should have. The *Deployment* handles how changes to the application are rolled out.

All these Kubernetes objects are specified by using either YAML or JavaScript Object Notation (JSON) and applied via the Kubernetes control plane by interacting with the Kubernetes API server. You can refer to the official [Kubernetes documentation](https://kubernetes.io/) for further information on Kubernetes.

**Serverless functions**

A given microservice of a cloud native application can be modeled as a *serverless function*. This programmatic function serves the business capability of a microservice that runs on a cloud infrastructure. With serverless functions, most of the management, networking, resiliency, scalability, and security are already being provided by the underlying serverless platform.

Serverless platforms such as AWS Lambda, Azure Functions, and Google Cloud Functions offer automatic scaling based on the load, support for multiple programming languages, and built-in features related to resilience communication, security, and observability. Microservices that need to support bursts of loads, batch jobs, and event-driven services are suitable to be implemented using serverless functions.

When you are using a serverless function, you may be using containers underneath, but that is transparent to the microservices developer. You can simply write a function with the business logic of your microservice and hand that over to the serverless platform to execute it. The details of how it is executed and deployed are also hidden away from the user.

**Virtual machines**

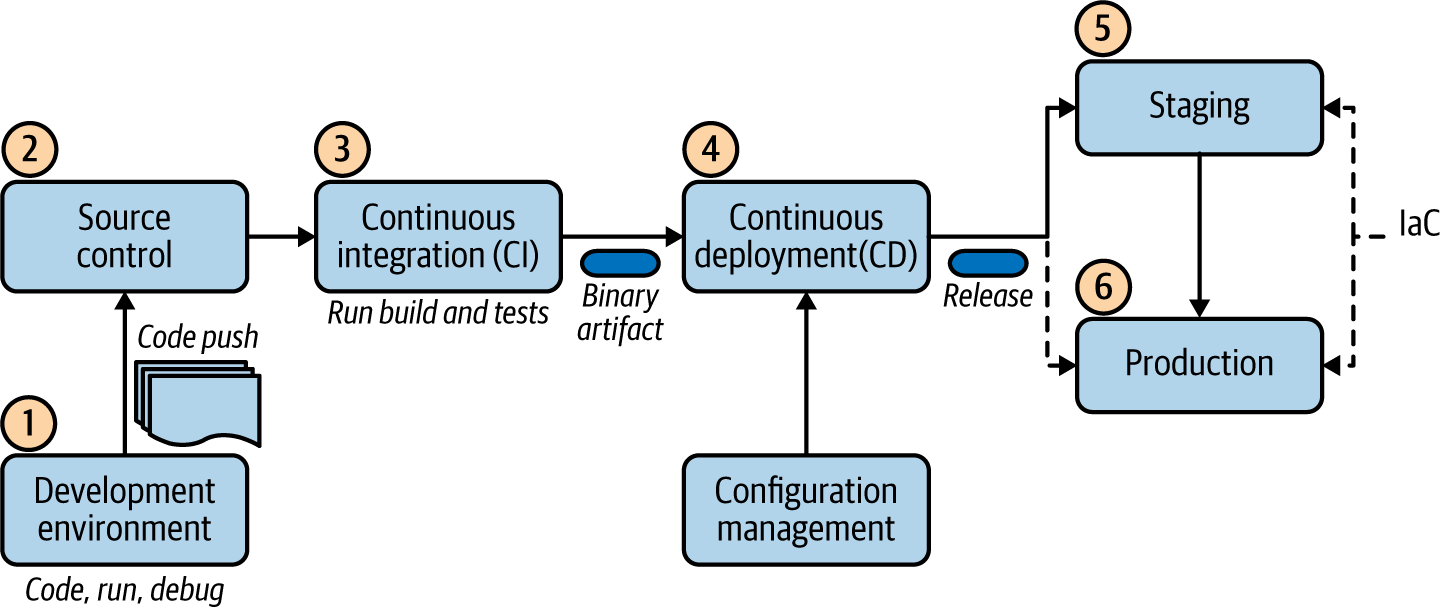
You may opt to run your microservices without using containers. While using containers to build cloud native applications is not mandatory, you need to manage the complexities and overhead of running applications on top of VMs. For this reason, in most real-world implementations of cloud native architecture, we often see the adoption of containers, container orchestrations, or higher-level abstractions such as serverless functions.

**Automate the Development Life Cycle**

When it comes to the delivery of cloud native applications, it’s important to be agile, quick, and safe. To achieve this, we need to streamline the entire life cycle of cloud native application development and automate every possible step.

*Automation* in the context of cloud native applications is all about automating the manual tasks of the development life cycle. This includes tasks such as running integration tests, builds, releases, configuration management, infrastructure management, and continuous integration (CI) and continuous delivery/deployment (CD).

In the development life cycle shown in [Figure 1-5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#cloud_native_application_development_li), you can see all the stages of building a cloud native application.



**Figure 1-5. Cloud native application development life cycle**

The development life cycle starts as the developers develop their code, and then run, debug, and push their changes into a central source-control repository such as Git. In the event of a code push, it automatically triggers the *continuous integration* process. This is where the code is built, the tests are executed, and the application gets packaged into a binary. A continuous integration tool automatically builds and runs unit tests on the new code changes to immediately surface any errors.

In deploying artifacts to different environments, the *continuous deployment* process kicks in. It picks the binary artifact that is built and applies an environment-specific configuration by using *configuration management* tools, and deploys the release to a specified environment. In this phase, we may run multiple parallel test stages before we push the changes to a production deployment. The final push to the production environment may be fully automated or may involve a manual approval step.

The difference between *continuous delivery* and *continuous deployment* is that in continuous delivery, manual approval is necessary to update to production. With continuous deployment, the update to production happens automatically, without explicit approval.

In automating the creation of the target environment (dev, staging, or production), the technique of *infrastructure as code* (*IaC*) is commonly used. With the IaC model, the management of infrastructure (networks, VMs, load balancers, and connection topology) is done using a declarative model that is similar to the source code of an application. With this model, we can continuously create the required environment by using the descriptor without any manual intervention. This improves the speed and efficiency of the development process while keeping the consistency and reduced management overhead. Therefore, IaC techniques are an integral part of the continuous delivery pipeline.

Once we define the designed state of the deployment, platforms such as Kubernetes can take care of maintaining that deployment state with the use of *reconciliation loops*. The key idea is to maintain the deployment state without any user intervention. For example, if we specify that a given application should run three replicas at a given time, Kubernetes reconciliation makes sure that three applications are running all the time.

**Dynamic Management**

When cloud native applications are deployed into a production environment, we need to manage and observe the behavior of the application. Here are some of the key capabilities needed to dynamically manage cloud native applications:

*Autoscaling*

Scales the application instances up or down based on the traffic or load

*High availability*

In the event of a failure, provides the ability to spawn new instances in the current data center or shift traffic to different data centers

*Resource optimization*

Ensures optimum use of resources, with dynamic scaling and no up-front costs but with real-time automated response to scaling needs

*Observability*

Enables logs, metrics, and tracing of the cloud native application with central control

*Quality of service (QoS)*

Enables end-to-end security, throttling, compliancy, and versioning across applications

*Central control plane*

Provides a central place to manage every aspect of the cloud native application

*Resource provisioning*

Manages resource allocations (CPU, memory, storage, network) for each application

*Multicloud support*

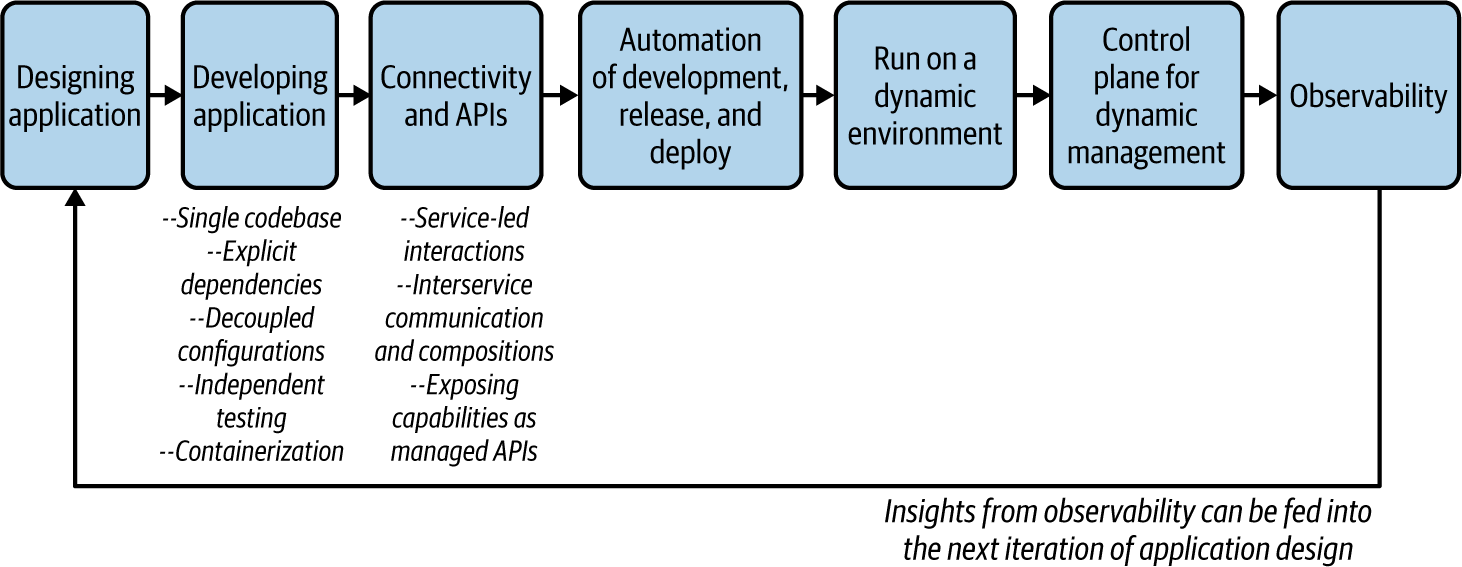
Provides the ability to manage and run the application across several cloud environments, including private, hybrid, and public clouds (as a given application may require components and services from multiple cloud providers)

Most capabilities of dynamic management are offered as part of popular cloud services such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). Containers and container orchestration systems such as Kubernetes play a major role in democratizing your applications across these cloud platforms so that your applications aren’t coupled to a specific vendor.

**Methodology for Building Cloud Native Apps**

Building cloud native applications requires you to follow a new development methodology, one that is different from the conventional approach that many of us have practiced. Some people believe the way to build cloud native applications is to use the [Twelve-Factor App](https://12factor.net/) methodology. However, we’ve found that this methodology has several gaps; it doesn’t cover every aspect of the cloud native application development life cycle.

Therefore, we have come up with a more complete and pragmatic methodology for building cloud native apps. We break this approach into phases and reuse some of the existing methodologies whenever necessary. [Figure 1-6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#methodology_for_building_id00204) illustrates the key phases of our methodology for building cloud native applications.



**Figure 1-6. Methodology for building cloud native applications**

Let’s dive into the details of each phase.

**Designing the Application**

When you are building a cloud native application that comprises microservices, you cannot just jump into the application development right away. You need to design the application around the business capabilities that you want to cater to. This requires you to clearly identify the business capabilities that the application has to offer as well as the external dependencies (services or systems) that the application needs to consume.

Therefore, in the design phase, you should have a closer look at the business use case and identify the microservices that you want to build. The designing of a cloud native application can use the domain-driven design (DDD) methodology, which builds abstractions over complex business logic and represents them in the software components.[**4**](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#idm46399014954776)

The DDD process starts with analyzing the business domain (e.g., retail or healthcare) and defining boundaries within that domain where a particular domain model applies. These are known as *bounded contexts.* For example, an organization might have bounded contexts such as sales, human resources (HR), support, and so on. Each bounded context can be further broken into *aggregates*, clusters of domain objects that can be treated as a single unit.

These bounded contexts may or may not be directly mapped to a microservice. When we are designing a cloud native application, typically we can start with a service for each bounded context and break it into smaller services that are built around aggregates as we progress. Once the DDD for the cloud native application is completed, you can also finalize the service interfaces/definitions and the communication styles while identifying the microservices.

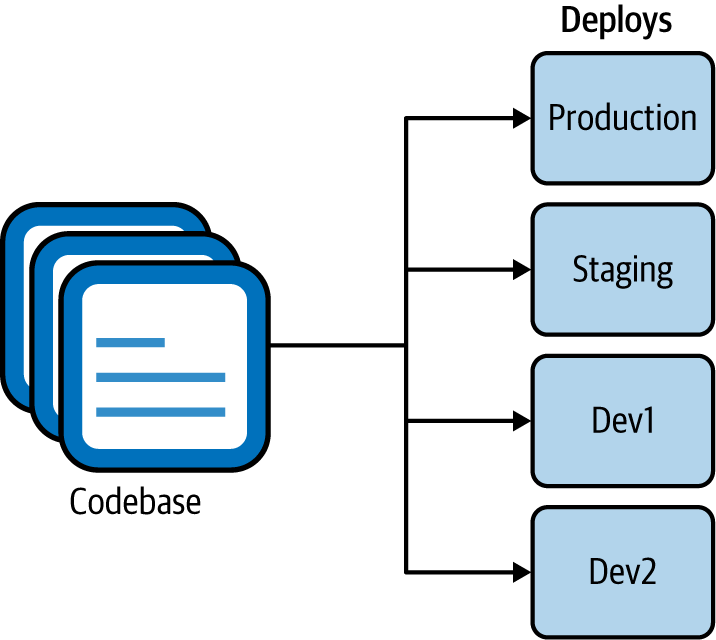
**Developing the Application**

In the development phase, we build the application based on the business use cases and service interfaces that we have identified in the design phase. In this section, we outline key aspects of the development process that enable cloud native applications.

**Independent codebase**

Each microservice of a cloud native application should have a *codebase* tracked on a version-control system (such as Git). Multiple instances of the service, known as *deploys*, will be running. So, as [Figure 1-7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#a_single_codebase_with_multiple_deploym) shows, you can deploy the service into different environments such as dev, staging, and production—all using the same codebase (but they may use different versions of the codebase).

Having an independent codebase means that the life cycle of the microservice can be completely independent from the rest of the system. And you can explicitly import external dependencies such as libraries.



**Figure 1-7. A single codebase with multiple deployments into different environments**

**Explicit dependencies**

All the code-level dependencies of a microservice must be explicitly declared and isolated from one another. The dependencies should be declared in a manifest that is part of the microservice code, and the service shouldn’t depend on any system-wide dependencies that are not declared explicitly.

**Decoupled configurations**

As we discussed earlier, cloud native applications contain a single codebase of a service that is deployed into multiple environments. This is possible only if the configuration of the microservice is fully decoupled from the microservice code. The codebase of a service is environment agnostic, and configuration varies among different deployments.

**Independent testing**

A microservice should have self-contained tests that independently verify its functionality. Usually, these tests are an integral part of the development cycle of the microservice, and verification of the microservice occurs during the build and deploy stages. We can consider these *unit tests* as they are localized to the scope of a given microservice.

However, because a cloud native application contains multiple microservices that work together to serve a certain business use case, unit tests alone can’t test the application’s overall functionality. We also need system-wide tests, known as *integration tests*. These tests collect microservices and other systems together and test them as a single unit in order to verify that they collaborate as intended, to achieve the larger business capability. You can find more details of microservice testing in [“Testing Strategies in a Microservice Architecture”](https://oreil.ly/rKa6u), by Toby Clemson.

**Containerization**

Most of the concepts that we discussed in previous steps can be demonstrated by containerization of the microservices that you build. While containerization is not mandatory to build cloud native applications, it is quite useful in implementing most of their characteristics and requirements.

Encapsulating a cloud native application into a single package with all the dependencies, runtimes, and configurations is enabled through containerization. Containerization (using technologies such as Docker) makes microservices *immutable*, which means they can be started or stopped at a moment’s notice and discard any faulty instances instead of being fixed or upgraded. This requires the microservices that we containerize to have a fast startup time and graceful shutdown times. Therefore, containerization works best when you leverage container native frameworks and technologies. (If fast startup time cannot be achieved because of an inherent limitation of the applications that we containerize, container orchestration systems such as Kubernetes provide readiness and liveness checks to ensure that the applications are ready to serve their consumers.)

When developing microservices, it is often required to connect with other microservices and/or expose business capabilities to external consumers as APIs. We cater to these requirements in the *next* phase, as we establish connectivity.

**Connectivity, Compositions, and APIs**

As we discussed at the beginning of this chapter, cloud native applications are distributed applications that are connected via network communication. As we design them as a collection of microservices, we often need to have interactions between those services and external systems. Therefore, having connectivity between the services and properly defining APIs and service interfaces is critical.

**Service-led interactions**

All microservices and applications should expose their capabilities as a *service*. Similarly, any external capabilities and resources that a microservice consumes should also be declared as a service (often known as a *backing service*).

The notion of a service is an abstraction that helps microservice interaction in many ways. A service is an enabler for dynamic service discovery, keeping a repository/registry of service metadata. It also allows you to implement concepts such as load balancing. That’s why the service abstraction is built into container orchestration platforms such as Kubernetes as a first-class abstraction. Therefore, when you build a cloud native application with a set of microservices, its capabilities can be declared as services (for example, a Kubernetes service). Any external application/service or resource (such as a database or message broker) that we consume should also be declared as a service that we can consume over the network.

**Interservice communication and compositions**

The interaction between services and other systems is a key part of the development of cloud native applications. These interactions happen over the network, using various communication patterns and protocols. These interactions may involve consuming multiple services, creating compositions, creating event-based consumers or producers, and so on. We also have to build certain features—such as application-level security, resilient communication (circuit breakers, retry logic with backoff and time-outs), routing, publishing metrics, and traces to observability tools—as part of the interservice communication logic, though they are not really part of the business logic. (We discuss interservice communication and composition in detail in Chapters [2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns), and [3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern), and [5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns).)

Therefore, as the service developer, you need to have the required capabilities in the technology stack that you use to build the services. Some of the commodity features that are not directly related to the business logic of the services (for example, resilient communication) can be implemented outside the application layer (often using the underlying runtime platforms such as the cloud provider that runs our applications). We’ll discuss all these patterns in detail in the upcoming chapters.

**Exposing capabilities as managed APIs**

For certain capabilities, the notion of a service may further extend into the concept of a *managed API*. Since most of the business capabilities of a cloud native application can be exposed to external and internal parties, we want to make it a managed service/API. This means you can use an API gateway and a management plane (API management/control plane) to implement capabilities (of the APIs that you expose to the consumers), such as security, throttling, caching, versioning, monetization (creating revenue from the APIs exposed), enabling a developer portal, and so on.

The *API gateway* acts as the front door to your capabilities, and a *developer portal* can nurture an ecosystem around your APIs. API management should be done for external as well as internal consumption of your services. However, API management is not built into container orchestration platforms such as Kubernetes. Therefore, you need to explicitly use API management technologies to expose your microservices as managed APIs.

**Automating the Development, Release, and Deployment**

As we noted previously in this chapter, automating as many steps as possible in the development, release, and delivery process is a vital part of building cloud native applications The various stages of building cloud native applications (such as testing, code push, build, integration tests, release, deployment, and running) should be automated by using continuous integration, continuous deployment, IaC, and continuous delivery techniques and frameworks.

**NOTE**

*Continuous Delivery* by Jez Humble and David Farley (Addison-Wesley Professional) is a great reference on how to implement a continuous delivery strategy for your software applications.

**Running in a Dynamic Environment**

In the running, or execution, phase of your cloud native application, you can set up the applications to be deployed and executed in an execution environment as part of the previous phase. The key idea here is to ensure that your application is independent from the execution environment and that it can be executed in various execution environments (dev, staging, production, etc.) without any changes to the application code. Since you use containers as the delivery model, the execution runtime often contains a container orchestration system. The execution environment can be a local environment; a public, hybrid, or private cloud; or even multiple cloud environments.

As Kubernetes is the most popular choice for container orchestration, we can use it as the universal runtime abstraction to deploy our applications so that their behavior will be similar across execution environments and multicloud scenarios. The dynamic nature of the environment—including container provisioning, resource management, immutability, and autoscaling—can be completely offloaded to Kubernetes. Also, as the container orchestration platform provides most of the dynamic-execution-related features, the application needs to worry only about capabilities that are within its scope (for example, scaling, concurrency requirements of a single runtime).

The orchestration platforms, such as Kubernetes, by default run your application as a stateless process (the state of the application is not maintained or persisted). However, if the application requires state, you have to explicitly use an external state store to keep the application state outside your application (such as in a data store) so that you can decouple the application state from the container life cycle. If you plan to run cloud native applications in a local data center or a private cloud, you can still benefit from Kubernetes, as it takes care of a lot of complexities of container orchestration.

**Control Plane for Dynamic Management**

In this phase, we use a central management and administration layer known as the *control plane* that allows you to control the behavior of the dynamic environments that your applications are running. This control plane is the main interaction point between the DevOps and developers who run their application in a runtime environment. Usually, such cloud control planes consist of a web interface as well as a representational state transfer (REST) or remote procedure call (RPC) API. Most cloud providers offer such control planes as part of their cloud service offerings.

**Observability and Monitoring**

Once you deploy and run your applications, the next phase of building cloud native applications is to observe their runtime behavior. *Observability*, in the context of a software application, refers to the ability to understand and explain a system’s state without deploying any new code. This is essential for troubleshooting, recording business transactions, identifying anomalies, identifying business patterns, generating insights, and so on.

In the observability and monitoring phase, you need to enable key observability aspects in your cloud native application. These include logging, metrics, tracing, and service visualization. Tools are explicitly built for each of these aspects, and most cloud providers offer these capabilities out of the box as managed cloud services. From the application-code level, you may have to enable agents or client libraries without changing your application’s code.

With that, we have discussed all the phases of the methodology for building cloud native applications.

**Design Patterns for Building Cloud Native Apps**

In the previous sections, we explored all the key characteristics of cloud native applications and the methodology for building them. As you have seen, cloud native architecture requires a significant change in the methodology, technology, and architecture for building software applications.

We cannot simply stick to the conventional design pattern of building software applications. Some patterns are becoming obsolete, others require certain changes or tweaks, and new patterns are emerging to serve the specific needs of cloud native architecture. These patterns can be applied at different stages of a cloud native application development life cycle. While the industry tends to focus on deployment and delivery of cloud native applications, the complexity of building the business logic, using various communication patterns, and connecting cloud native applications has often been overlooked.

In this book, we focus on the design patterns that you can use when building cloud native applications. These are the patterns that you have to apply when building the business logic of cloud native applications, connecting them, and enabling external parties to consume them. Depending on the nature of the cloud native application and the patterns you use to build it, the cross-cutting capabilities such as deployment, scaling, security, and observability may also be implemented differently. We discuss those capabilities from the perspective of cloud native application development and dive into them whenever required.

In the following chapters, we examine patterns in the context of six key areas: *communication*, *connectivity and composition*, *data management*, *event-driven architecture*, *stream processing*, and *API management and consumption.* Let’s briefly summarize each one.

**Communication Patterns**

As you have learned, a cloud native application is composed of a collection of microservices, distributed across a network. The cloud native *communication patterns* are all about how these services can communicate both with each other and with external entities.

To build even a very simple business use case, your application needs to consume external services (which could be another service, a database, or a message broker, for example). Therefore, building the interaction between your application and these external services is becoming one of the most common and yet most complex tasks in building cloud native applications.

Most of the conventional interservice communication patterns and technologies of the distributed computing world are not directly applicable in the context of cloud native application development. We need to select communication patterns that are well suited for cloud native attributes of the application (for example, patterns that allow service autonomy and scalability) as well as the business use case (for example, some may require delivery guarantees, while others may require real-time responses).

The interservice communication among cloud native applications is implemented using either synchronous or asynchronous communication patterns. In *synchronous communication*, we use patterns such as request/response and RPC. In *asynchronous communication*, we use patterns such as queue-based and publisher-subscriber (pub-sub) messaging. In most real-world use cases, you need to use both categories together to build the service interactions. Service interface definitions and contracts also play a vital role when it comes to communication patterns, as they’re the standard way of expressing how a given service can be consumed.

In addition to the service-to-service interactions, certain cloud native applications may have to communicate with external parties such as frontend clients or backing services. As an application developer, you need to work with a lot of moving parts and a lot of interactions with external services and systems.

In [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns), we discuss all these communication patterns in detail, along with the related implementation technologies and protocols.

**Connectivity and Composition Patterns**

The more microservices you have, the more interservice communication will take place. Therefore, when you design cloud native applications, you need to bring in certain capabilities and abstractions that reduce the complexity of interservice communication. That’s where the *connectivity and composition patterns* come into the picture.

**Connectivity**

In the context of interservice communication, *connectivity* refers to establishing a reliable, secure, discoverable, manageable, and observable communication channel among services. For example, when a given service calls another service, you need to apply certain reliability patterns such as retrying or establishing a secure communication channel. They are not part of the business logic of the application but are essential to building strong connectivity.

In [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern), we discuss various patterns related to resilient communication, security, service discovery, traffic routing, and observability in interservice communication. We’ll also explore how interservice connectivity infrastructures such as a service mesh and sidecar architecture facilitate these requirements.

**Compositions**

When building cloud native applications, it’s quite common to create a service by plumbing, or integrating, one or more other services or systems. These are known as *compositions* (also known as *composite services* and *integration services*).

As we discussed at the beginning of the chapter, services and systems were often built using SOA before the cloud native era. In SOA, all the services, data, and systems are integrated using an ESB—so when creating compositions, ESB was the default choice. A plethora of composition patterns were used in this architecture, which were commonly known as *enterprise integration patterns* (*EIPs*).

However, in the cloud native era, we don’t use a central composition layer. All such tasks need to be done as part of the services we develop. Therefore, in [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern), we dive into all those composition patterns and identify which ones we should apply to building cloud native applications.

**Data Management Patterns**

Most cloud native applications that you develop need to take care of some data management. Your application is often backed by a database that acts as persistent storage to store the application state or the business data required to build the service. As you learned previously, cloud native applications are inherently distributed. Hence, data management is also done in a completely decentralized way.

In conventional monolithic applications, we used to have a central, shared data store, with which many applications interacted. With cloud native applications, we let a given microservice own its data store, and external parties can interact with it only via that service interface. With this segregated data management approach, accessing, sharing, and synchronizing data among microservices becomes challenging. That’s why knowing the cloud native data management patterns is essential for cloud native application development.

In [Chapter 4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch04.html#data_management_patterns), we explore a wide range of cloud native data management patterns covering decentralized data management, data composition, data scaling, data store implementations, handling transactions, and caching.

**Event-Driven Architecture Patterns**

When we discussed cloud native communication patterns, we discussed asynchronous messaging as an interservice communication technique. That is the foundation of event-driven cloud native applications. *Event-driven architecture* (*EDA*) has been widely used in application development for decades. In the context of cloud native applications, EDA plays a vital role, as it’s a great way to enable autonomous microservices. Unlike synchronous communication techniques such as querying or RPC, EDA enables more decoupled microservice interactions.

Therefore, we dedicate [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns) to exploring most of the commonly used patterns in EDA and how to leverage them for building cloud native applications. We cover various aspects of cloud native EDA, including event delivery patterns (queue-based, pub-sub), delivery semantics and reliability, event schemas, and related implementation technologies and protocols.

**Stream-Processing Patterns**

In EDAs, we deal with a single event at a time. In other words, the microservice’s business logic is written to deal with a single event at a time. There’s no correlation between subsequent events. A *stream*, on the other hand, is a sequence of events or data elements made available over time. Those events are processed by the application in a stateful manner.

The implementation and deployment architecture of such a microservice changes drastically from an event-driven microservice because it has to handle state, do efficient data processing, manage various scaling and concurrency semantics, and so on. That’s why we have dedicated [Chapter 6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#stream_processing_patterns-id00204) to stream-based cloud native patterns.

The notion of building application logic to process or produce such a stream is commonly known as *stream processing*. Building cloud native applications by using stream-based architecture is becoming common, as it enables the microservices to process massive continuous data streams statefully.

**API Management and Consumption Patterns**

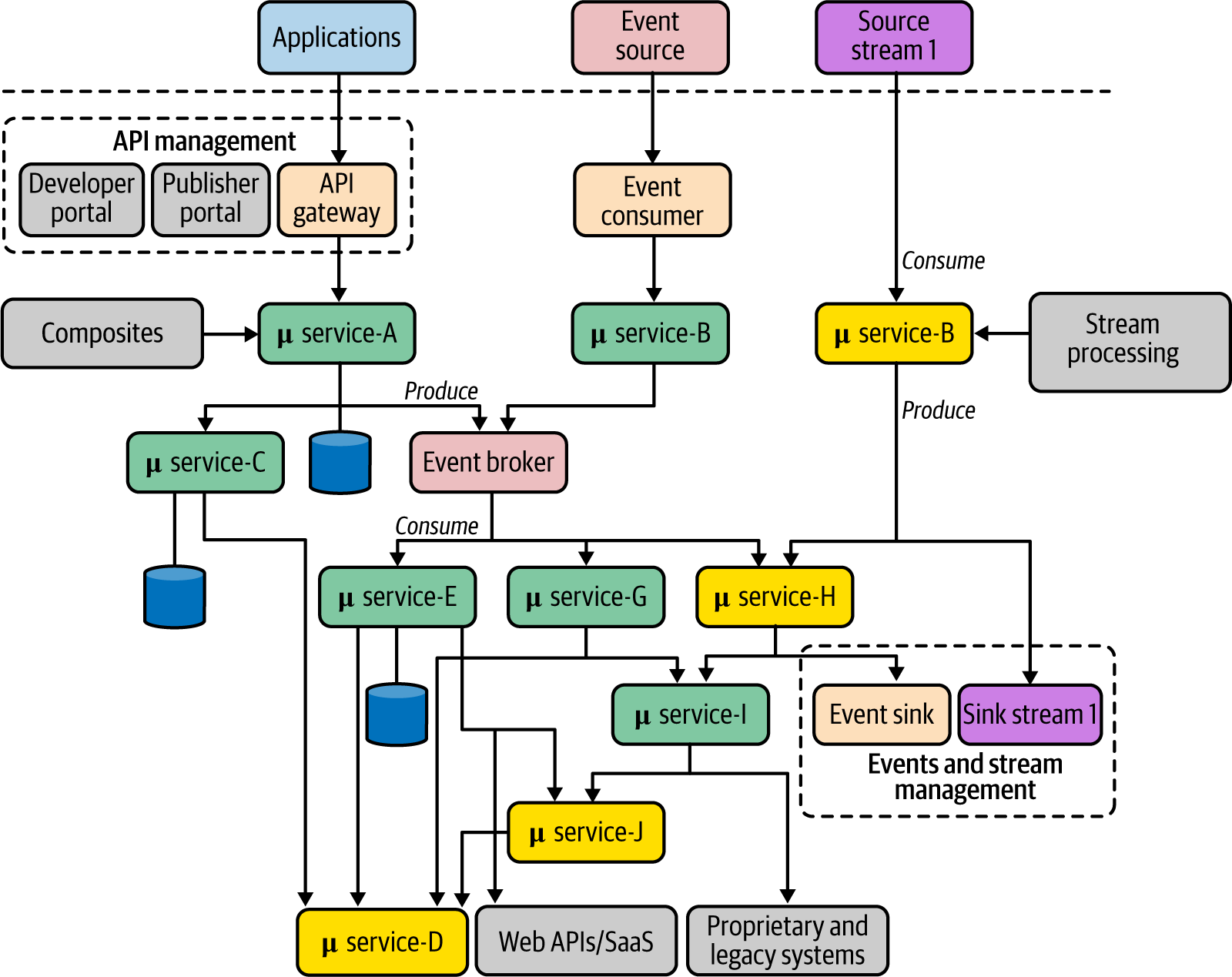
In most medium or large-scale use cases of cloud native architecture, you have to expose certain business capabilities of your applications to the external or internal parties that are outside your application scope. You need to expose such capabilities as managed services or APIs. This allows you more control over how external parties consume those capabilities, and enables external parties to easily discover and provide feedback on those APIs.

Exposing these capabilities is often done by using a separate API gateway layer that acts as the front door to all the APIs that you expose. The API gateway also includes a management plane and developer portal that are built around the APIs exposed. [Chapter 7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_and_consumption_patterns) covers several patterns related to API management and consumption.

Now that you’ve learned the foundational concepts of cloud native application development, let’s place those concepts into a reference model so you can understand how they are used in a real-world cloud native application architecture.

**Reference Architecture for Cloud Native Apps**

In most real-world cloud native applications, we commonly see a combination of development strategies. [Figure 1-8](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#a_generalized_architecture_for_building) shows these various strategies in a generalized architecture. This reference architecture comprises multiple microservices that are communicating with different communication patterns. Each service may use its own data or persistent store, and there is a shared or private event broker infrastructure as well. The interaction among microservices represents all the communication patterns that we can implement. Each communication link can be implemented by using connectivity patterns related to reliability, security, routing, and so on.



**Figure 1-8. A generalized architecture for building cloud native applications with APIs, events, and streams**

As you can see, microservices are creating composites out of one or more services, such as microservices A, E, and G. Such services are built using various composition patterns. When that application needs to be exposed as a managed business capability outside your application’s realm, you can leverage API management. All the external applications can consume those capabilities via an API gateway, and you manage it via the other components of the API management layer. For the services that are based on EDA, often it is essential to have an event broker solution in place. Services may use a shared broker as the simple eventing infrastructure or can have their private event brokers as well.

Stream-processing services follow a similar approach, but the stream-processing logic may be implemented using a drastically different set of patterns and technologies. Both event- and stream-based services feature event/stream management for the producer (sink) side of the service.

This reference architecture may look complex at first glance, but in the upcoming chapters, we dive into every aspect and explore the patterns, implementation technologies, and protocols that you can use to realize it.

**Summary**

Cloud native is a modern architectural style that empowers organizations to build agile, reliable, safe, scalable, and manageable delivery of software applications. Cloud native is a way of building software applications as a collection of loosely coupled, business-capability-oriented services that can run in dynamic environments in an automated, scalable, resilient, manageable, and observable way.

Cloud native applications are designed as a collection of microservices, packaged into containers and managed with container orchestration systems such as Kubernetes, automated with CI/CD, and managed and observed in a dynamic environment. By considering all these characteristics, we can use a complete and pragmatic methodology for building cloud native apps that covers design, development, interconnectivity, API management, and execution and management in a dynamic environment.

We can apply a wide range of design patterns when building cloud native applications. In this book, we focus mainly on the development patterns that you have to apply when building the business logic of cloud native applications, connecting them, and enabling external parties to consume them. We discuss these patterns under six key areas: communication, connectivity and composition, data management, event-driven architecture, stream processing, and API management and consumption. In the next chapter, we dive into cloud native communication patterns.