**7. Serverless Architecture**

Shivakumar R Goniwada[**1**](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_7_Chapter.xhtml#Aff2)

(1)

Bangalore, India

In the previous chapters, I discussed microservices and event-driven concepts and their architecture and use cases. In this chapter, I will explain serverless and function as a service (FaaS) and show how they are useful for your enterprises, and I will present some relevant use cases.

Serverless architecture allows the rapid development of cloud native applications that can handle various levels of traffic. The term *serverless* refers to the fact that the cloud provider provides a service without requiring you to manage or administer servers. Your code is executed on demand, as it is needed. You do not need to deal with physical servers, and the complexity of how compute resources are provided is hidden from you.

Serverless is a way to define the service, practices, and strategies to be agility in your development so you can embrace innovation and respond faster to business disruptions. With serverless computing, you don’t need to worry about capacity management, infrastructure management, and so on.

Serverless architecture is still maturing, and all the major cloud providers offer various serverless services, including FaaS, databases, IoT services, and more.

In this chapter, I will cover these details of serverless architectures:

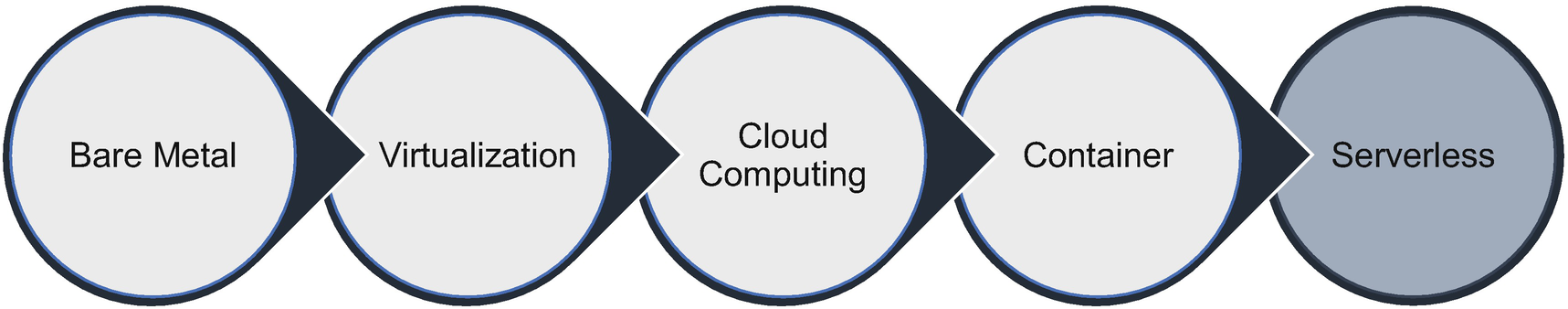
* Introduction to serverless computing and architecture
* Usage of serverless
* Journey to serverless architecture
* Relevant use cases
* The cost benefits of serverless
* Serverless in various cloud providers

**Evolution of Serverless**

So that you can understand serverless better, I will explain how servers have evolved in the context of serverless. A more thorough history of servers in general has already been provided.

As shown in Figure [7-1](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_7_Chapter.xhtml#Fig1), the deployment process has had several evolutions:

* Bare-metal technology is a large physical server with a single operating system.
* Virtualization technology virtualizes bare-metal servers into individual virtual machine resources.
* Each virtual machine is subdivided into containers.
* A service built on containers requires a function to run on managed services, called *serverless*.



***Figure 7-1***

Evolution of serverless

The evolution of IT architecture has been driven by a series of technological innovations. In this process, the resources are broken down, operational efficiency is increased, and management of software is simplified. The innovation of IT architecture has followed a few characteristics:

* Hardware resources become more granular.
* Resource utilization increases.
* Operation management is gradually reduced.
* Automation and intelligence are increased.
* IT is more focused on solving business problems.

Enterprises are already getting the benefits of VMs and containers in a cloud including cost savings, operationalization, improved agility, etc. Although the cloud eliminates the need for enterprises to manage their own data center, any server-based architecture still requires enterprises to architect for scalability, high availability, reliability, fault tolerance, etc., and companies need to take responsibility for patching and deployment.

Serverless is designed to address these challenges by providing enterprises with a different way of approaching application design. It eliminates the complexity of managing the servers at all levels of the technology stack and implements effective pay-per request billing models.

**What Is Serverless Computing?**

Serverless is a way of describing the services, practices, and strategies that enable you to adopt agility in your development so you can embrace innovation culture and move faster to market.

Serverless computing is a method of providing back-end services on an as-used basis, and it allows you to write and deploy programs without worrying about infrastructure management.

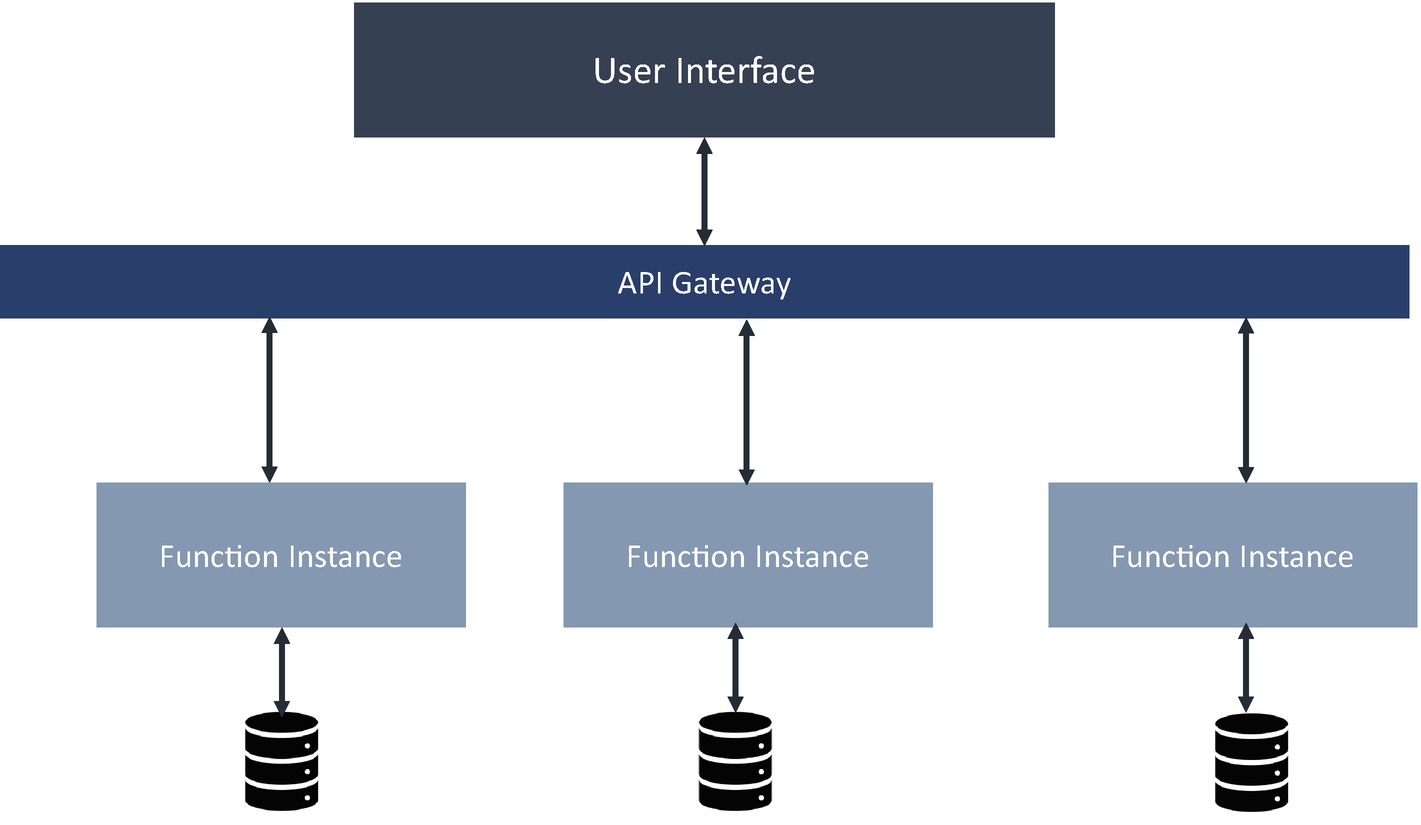
The demand for serverless technologies is increased because it provides an opportunity for faster time to market by dynamically allocating the required compute and memory based on load.

Serverless services have built-in autoscaling, high availability, and pay-as-you-go billing models. This provides a cost savings through infrastructure management, which enables enterprises to use an IT budget for innovation and upskilling. The pay-as-you-go model with serverless technologies shifts from having capital expenditures to flexible on-demand consumption, allowing users to scale, customize, and provision computing resources as and when required.

As explained earlier in the book, for a traditional architecture, if you want to develop a web application or any software, you had to own the physical hardware, with it managed by you or managed a third party to run a server. This required a lot of management and resources, and in the end, it was expensive.

Serverless technologies effectively shift the operation management from you to the cloud provider. With a serverless operational model, there are no servers to provision, patch, or manage, and there is no management of software such as installation, operations, etc. Many enterprises that have embraced serverless as mainstream are adopting more frequent releases of services, resulting in a faster time to market.

Serverless architecture can be used for many types of software services. Some of the common types of systems that are suited to a serverless architecture include web applications, event-driven data processing, event workflows, scheduled tasks, mobile applications, chatbots, and IoT systems. Figure [7-2](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_7_Chapter.xhtml#Fig2) shows an example of a system that has a serverless architecture.



***Figure 7-2***

Serverless computing architecture

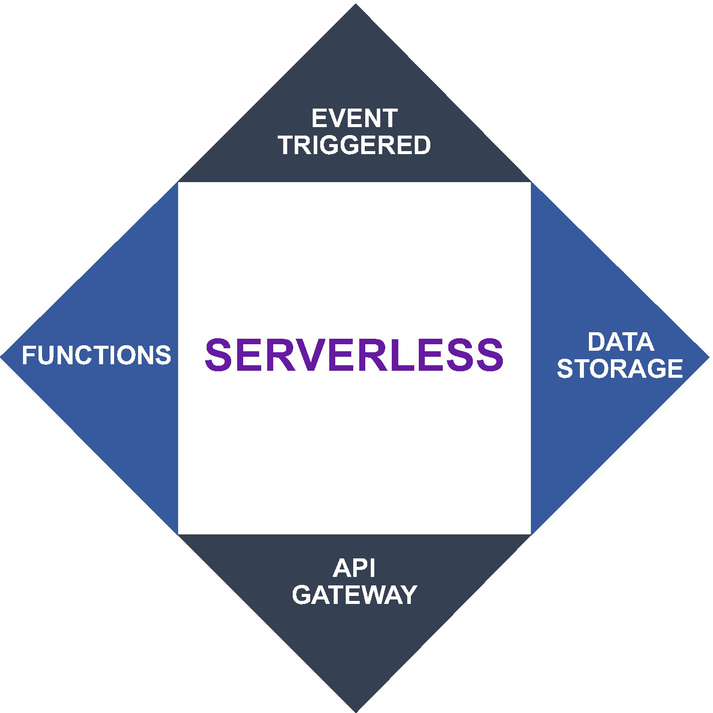
Serverless computing has three main benefits.

* *No server management*: There is no operating system to install or patch.
* *Flexible scaling*: Scale is managed for you or is done in a way that’s defined in terms of the actual capacity of the application as opposed to having to consider things such as CPUs and memory and other kinds of server-based concepts.
* *Automated high availability*: All the serverless components of the overall platform have built-in high availability. You don’t need to design for HA. Serverless gets HA out of the box.

To summarize, serverless computing still has the virtualized, containerized services underneath it, but people don’t interact with those servers anymore. All those infrastructure tasks, such as provisioning, scaling, and cleaning up, are done by machines in a completely automated lifecycle.

**Essential Components of Serverless**

Serverless architectures dramatically simplify the development of event-driven applications and microservices by removing the need to develop and maintain many complex architectural resources such as service meshes, event meshes, dynamic discovery systems, load balancers, retries, circuit breakers, bulkheads, topics, and queues. Instead, you need to focus on four essential components, as shown in Figure [7-3](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_7_Chapter.xhtml#Fig3).



***Figure 7-3***

Elements of serverless

**Event Triggered**

The applications provide a constant stream of events to which functions subscribe and act upon. The details of the event are stored, and a trigger for the function occurs. Examples are triggers, orders placed, comments posted, setting changes, images uploaded, etc.

**Data Storage**

Storage ranges from simple disk storage to highly scalable data stores. Storage is always provided as software as a service (SaaS) by the cloud provider. Examples are details stored, state stored, etc.

**Functions**

Functions are the actual business logic that is spawned solely to process an event. As events come in, they are processed based on the request.

**API Gateway**

A light API gateway provided by the cloud platform may be in the form of streaming or REST calls to support endpoints for mobile devices. Examples are outcomes, notifications created, photos shared, analytics generated, etc.

**Serverless and Event-Driven Computing**

The event-driven architecture (EDA) of serverless computing means that every component is independent and decoupled. The ability to listen to events and react to them once they happen in an elastic, scalable manner is a key advantage of serverless.

One of the key benefits of an event-driven approach in a serverless architecture will help to eliminate waste from an actual infrastructure perspective. Most servers in cloud services are in data centers, waiting idle, listening to a port waiting for asynchronous requests to come in, or waiting for an event to happen. Whether your server is active or idle, you still need to pay for it.

With serverless, the whole model is turned on its head to where now you are paying only for the compute time that you’re using. So, if your application goes through periods of low utilization and periods of high utilization, as an engineer you only need to worry about how to handle each event. Then your infrastructure, including your actual costs to run that application, accrue only when the code itself is running in response to those events.

**Serverless Design Principles**

Serverless eliminates a lot of common application architecture problems. When developing a serverless system, it’s more important to follow modern-day architecture principles along with related serverless architecture principles. Several principles can be used to design your application with serverless architecture. These define what serverless architecture looks like and what the properties exhibit. However, serverless makes it easy for developers to design and develop applications more readily. This reduces a lot of operational overhead by adhering to some of the 12 Factor App principles that are inherited when adopting serverless frameworks. The following sections provide you with more clarity when designing an application using serverless architecture.

**Stateless Functions**

FaaS is ephemeral; hence, you can’t store anything in memory because the compute containers running your code will automatically be created and destroyed by your application. Stateless is good to scale applications horizontally. Follow the single responsibility principle and only write functions that have a single responsibility and follow the right granularity. The appropriate granularity should be decided based on requirements and context.

**Push-Based and Event-Driven Pipelines**

Create push-based, event-driven pipelines to carry out complex computations and tasks. Use a choreography approach to interact between various serverless functions and try to build a way to create event-driven pipelines. Avoid polling or manual intervention where possible.

**Config: Store Config in the Environment**

FaaS providers such as AWS Lambda, Azure Functions, and Google Cloud Functions have separate environment variable sections where you can configure the key-value pairs that are made available to your functions at runtime. These configs can include resource handles, credentials, or environment-specific variables. So, by design, FaaS providers allow you to separate these from the code and eliminate the need to use any heavy frameworks.

**Backing Services: Treat Backing Services as Attached Resources**

FaaS frameworks inherently provide a clear separation between the business function code and the resources it accesses through the network. The framework does not allow you to run a dependent service or multiple processes inside the function.

**Concurrency: Scaling Out via the Process Model**

FaaS framework scaling comes out of the box as FaaS frameworks are designed to automatically scale in and out to meet the demands.

**Disposability: Maximize Robustness with Quick Startup and Shutdown**

FaaS frameworks are ephemeral. A container cannot be reused, so when a new function starts, it is not possible to start quickly, and there is some latency about setting up the execution context and bootstrapping. You need to follow best practices to keep your container warm. Usually, when you invoke frameworks the first time, it does download the dependencies, creating a container, and it starts the application before executing the code. The whole duration is known as the cold-start time. Once the container is up and running, for subsequent function invocation, the framework is already initialized, and it just needs to execute the function logic, called the warm-start time. You need to make sure you choose the appropriate steps to be the warm-start time.

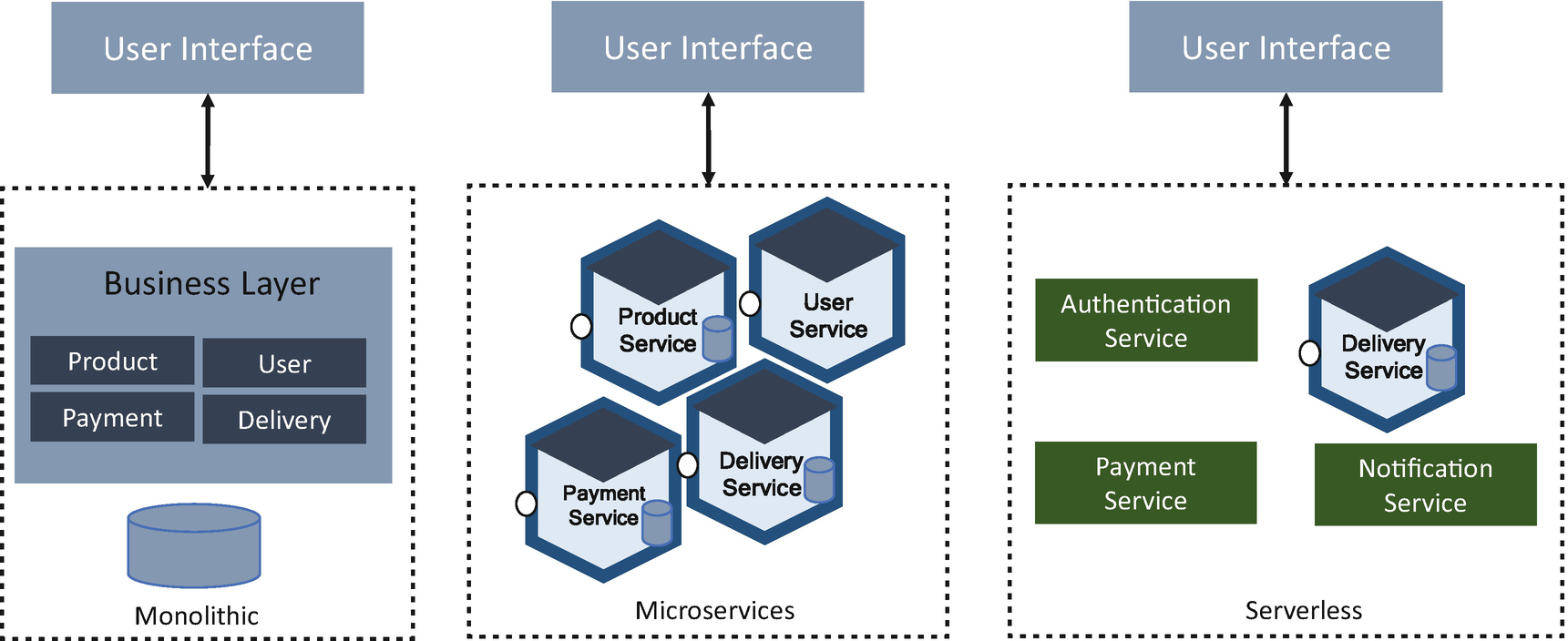
**Key Considerations for Serverless Computing**

When you are considering the serverless platform, you need a culture of cloud and a culture of automation, and you need to embrace a nimble architecture. The following are a few guidelines that will help you to take full advantage of serverless computing:

* **Leverage the entire platform in the cloud appropriately.**
* The best result occurs when an enterprise starts by identifying the use cases that will optimize the dynamic serverless platform. To help enterprises achieve this goal, cloud providers offer a rich set of tools that integrate effectively with the serverless platform. A large independent software provider like open source or commercial software (COTS) also provides deployment, monitoring, and storage solutions with great support for serverless applications on the cloud. These include NoSQL database partners such as MongoDB, Atlas, Couchbase, etc.; continuous integration (CI) and continuous delivery (CD) partners such as CodeShip and CloudBees for managing automated deployment pipelines; and monitoring partners such as Dynatrace, SignalFx, and I/O pipes with deep integration available for serverless platforms like AWS Lambda, Azure Functions, Google Cloud Functions, etc.
* **Don’t try to reuse existing application code in a serverless environment.**
* You may have a ready application that was developed earlier and would like to repurpose it. But if you do that, you’ll end up having way more code and heft than what the serverless environment is designed for. The suggestion is to refactor your code to adopt serverless characteristics such as stateless, function, ephemeral, etc.
* **Use existing platform components from providers for nonbusiness logic application functionality.**
* Serverless computing already has all the execution components you need already designed to work with serverless computing. For anything that is not related to business logic, the platform components always are your first choice; otherwise, there is no point in using serverless.
* **Create a reference architecture to guide all application development.**
* By creating a solution blueprint for what the system should look like, you ensure consistency across your application. This helps to create standardization and socialization across teams and resources, and it avoids mistakes.
* **Support DevOps.**
* Your development must create a DevSecOps pipeline before you begin your process.
* **Make a culture shift.**
* In my experience, you may need to be ready to do as much work restructuring organization and culture as you devote to the building of serverless applications.
* **Be idempotent.**
* Functions should be designed to be idempotent so that multiple executions of the same request yield the same result, and if the same request is processed more than once, there should be not any adverse effects.
* **Use an API gateway.**
* An important part of a serverless architecture is its API gateway. An API gateway is the entry point to a system. It is an HTTP server that takes requests from clients and routes to the relevant function containers.

**Why Use Serverless Architecture?**

The way you develop applications has changed dramatically in the past few years. As shown in Figure [7-4](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_7_Chapter.xhtml#Fig4), in a legacy monolith architecture, you host a single application on the server. A few years back, the microservice architecture style started to evolve. In microservices, each component is an independent service that can scale horizontally and solves many problems of the day-to-day business. In a microservices architecture, you need to provision and configure the infrastructure. Serverless almost behaves the same as microservices, but you do not need to worry about any runtime environment or deployment environment.



***Figure 7-4***

Monolith/microservices/serverless

Hosting a software application in a production environment involves a few parts. These parts are as follows:

* Code that solves a business problem
* Development and testing pipeline that automates the code build and testing lifecycle
* Deployment and infrastructure service provision, which creates containers, uses Kubernetes or provisions VMs, and configures the infrastructure and required software and runtime environment for your code
* Deployment of your code to a provisioned environment

Using an infrastructure from a cloud provider eliminates the physical hardware concerns but still requires the management of software installation and patches.

With a serverless architecture, you focus only on the code that solves the business problem, and serverless will take care of the rest of these points. This reduces the overhead of the agile pod team.

**Best Practices of Serverless Architecture**

After you build a serverless architecture, how do you grow an application and manage your source code repository for serverless? Usually, moving to serverless starts with FaaS and then you use the inherent functionality of scalability, etc., and move into production. Your application will grow more complex over time, performing multiple tasks to handle increasingly complex business logic, which leads to the following:

* A tightly coupled codebase
* Slower release cadence
* Poor discoverability
* Additional complexity
* Difficult to separate responsibility of ownership
* Difficult to maintain

Even though you added more functionality into an existing function, that will not be an issue. It still runs and provides the required output. The problem is that the function becomes monolithic, and you might face problems debugging and decoupling the function. To overcome all these traps, you need to adopt the following best practices:

* Design your function as an independent reusable function.
  + Each function is independent.
  + This reduces the impact of bugs or failures in one area of the code that affects the operation of other applications.
  + Allow a function to scale independently according to demand.
  + Share the logic, not the function, if two microservices access the same function.
  + Use contract testing and versioning.
  + Maintain a single source for libraries and dependencies for easier patching and updates in a serverless framework.
* Design your services for choreography.
* Organize source code repositories.
  + Group common functionality together for ownership and repositories.
  + Create repos around a group of functions and resources.
* Design your services with zero trust security.
  + Trust “no one” by default.
  + Build microperimeters around each resource to enforce strict verification of every person or service.
  + Execution role permissions can be limited by the application’s permission boundaries.
  + Monitor for insecure flows and attempt to force a function into an unsafe code path.
  + Store secrets in a secret manager, not in environment variables.
* Do not reinvent the wheel; use already proven services in the cloud.
* Use custom resources to enable cross-account service referencing.

**Types of Serverless Architecture**

Serverless architecture is a way to build and run applications and services without having to manage infrastructure. Your application runs on servers, but all the management is done by the providers. Serverless is focused on any service category, be it compute, storage, database, messaging, API gateway, etc.

Serverless has two similar operational attributes that are frequently used together. The major cloud providers include both.

* Function as a service
* Backend as a service

**Function as a Service**

FaaS is a serverless way to execute a modular piece of code in a self-managed container and is focused on event-driven computing. It is a paradigm wherein a function is a computation that takes some input and produces some output. FaaS gives you a fast way to focus on building cloud native microservices by abstracting away the complexities of managing virtual machines or clusters of containers.

FaaS can be accessed through events or APIs that you define when you create a function. There are many functions in your architecture where you deploy outside of your service, and that service can be accessed through an event and API. Here are some examples:

* A function can take an input of an image and output a label of that image.
* A function can take a notification and output a personalized email.
* A function can take a YouTube video URL and output a statistic of that video.
* A function can take income numbers and output a total tax calculation.

Various cloud vendors provide the FaaS architecture, covered in the following sections.

**AWS Lambda**

According to Amazon, “AWS Lambda is a serverless compute service that lets you run code without provisioning or managing servers, creating workload-aware cluster scaling logic, maintaining event integrations, or managing runtimes. With Lambda, you can run code for virtually any type of application—all with zero administration.” Lambda supports Node.js, Python, Go, Java, Ruby, and .NET.

You need to use AWS Identity and Access Management (IAM) to manage security in Lambda. For account-related tasks, you can manage permissions to access Lambda functions in the permission policy with users, groups, and roles.

You can call Lambda functions synchronously or asynchronously, with synchronous invocation. It is an I/O blocking service and will wait for the function to process the event and return a response. In the asynchronous invocation, it is non-I/O blocking. Lambda queues the event for processing and returns a response immediately. In cloud native architecture, I suggest going with an asynchronous invocation.

In serverless architecture, some use cases require a stateful nature and require running at the edge location. These are two features where the Lambda team is working.

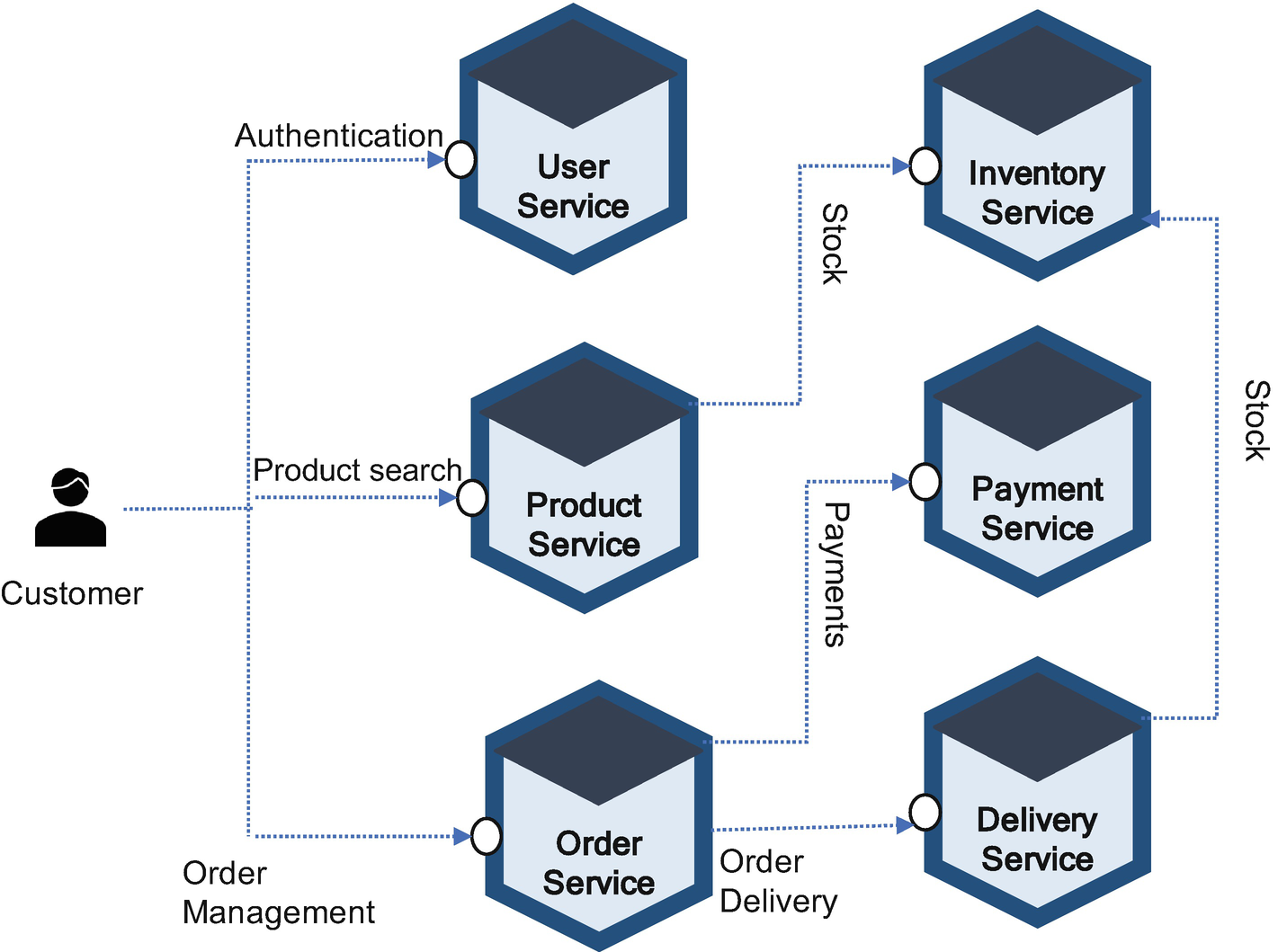
**Reference Architecture**

The following section provides a few Lambda reference architectures; these reference architectures are just an example to show how Lambda can be used.

**Ecommerce Reference Architecture**

I will use the same ecommerce architecture as illustrated in the previous sections and will show how you can leverage Lambda for this. In the abstract version of Figure [7-5](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_7_Chapter.xhtml#Fig5) of the ecommerce application (which included only the main services), I considered the following services:

* *User service*: Provides user management, authentication, and authorization
* *Product service*: Product information

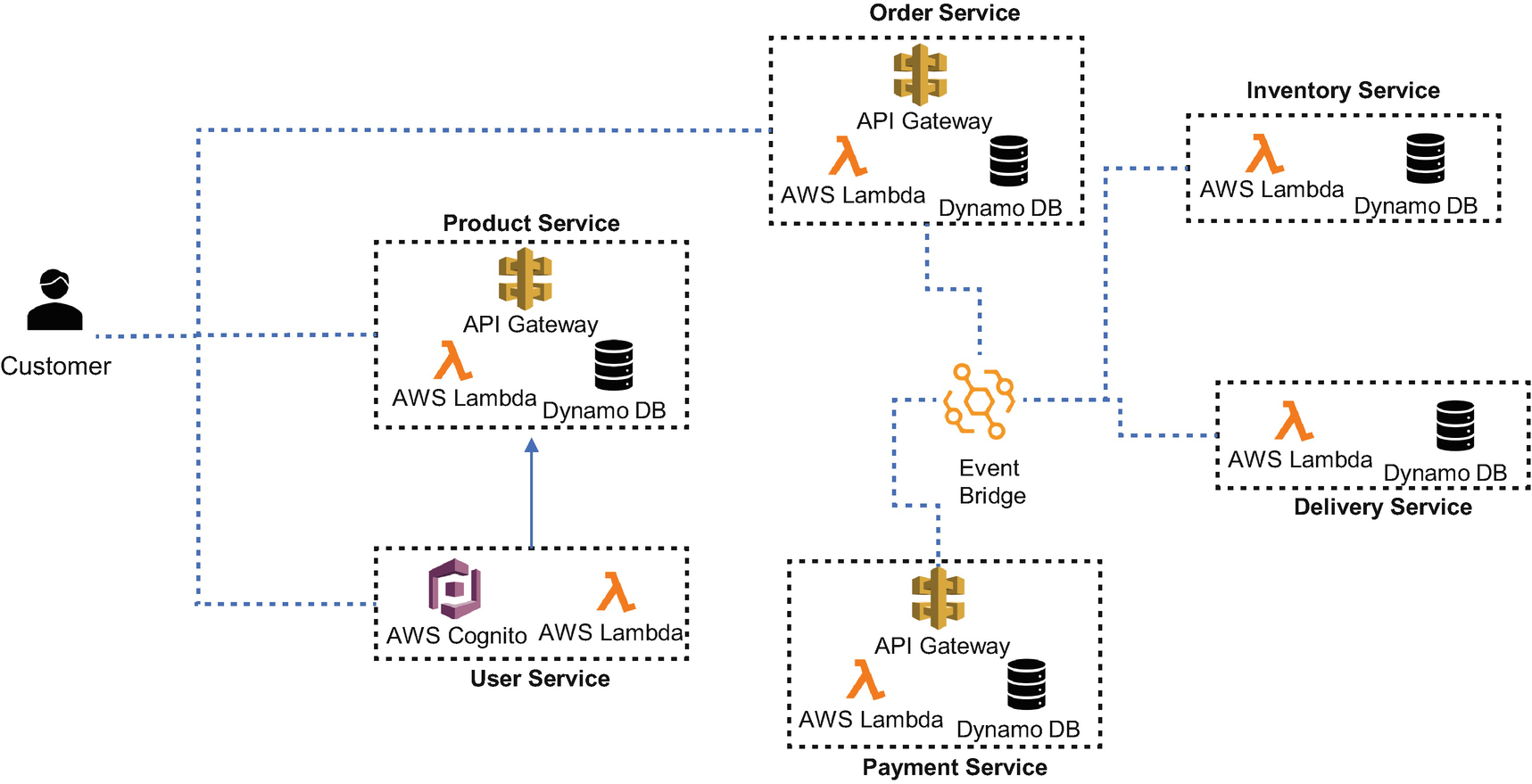


***Figure 7-5***

Ecommerce system services

* *Order service*: Manages the order creation
* *Inventory service*: Manages the stock management
* *Payment service*: Manages payment collection and refunds
* *Delivery service*: Manages shipping and tracking

Figure [7-6](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_7_Chapter.xhtml#Fig6) illustrates how you can use the Lambda function along with other AWS services of these ecommerce services. Here I am not showing the entire architecture that includes data synchronization, third-party payment services, etc., and showing only high-level services and interaction across Lambda functions.



***Figure 7-6***

Referencing serverless architecture for AWS Lambda

* *API gateway*: For service-to-service synchronous communication and access from mobile and web applications, you can use any API gateways like Layer 7, Apigee, etc.
* *Event bridge*: For service-to-service asynchronous communication, you can use any event broker like Kafka, etc.
* *AWS Cognito*: For managing and authenticating users, this provides the JSON web tokens used by web services.
* *AWS Lambda*: The service implementation communicates with APIs and events.
* *AWS DynamoDB*: For storage service, you can use any service like MongoDB, Couchbase, Cassandra, etc.

**Best Practices of Lambda**

Here are some best practices of Lambda:

* *High availability*: By default, Lambda accesses S3 and DynamoDB in a VPC and won’t access RDS, Elasticsearch, etc. To make these available to Lambda, select multiple subnets in different availability zones.
* *Concurrency*: Lambda handles scalability on its own. Lambda can scale up to 1,000 instances per region as of writing this book. You should always consider the limitations of other integrated services so that you can adjust the concurrency limit for the function based on the maximum number of connections these services can handle
* *Throttling*: The concurrency is limited to 1,000 requests as of writing this book. If a request exceeds 1000 requests, how will you handle your function? If the concurrency is synchronous, your service receives an error code with error details, and if the concurrency is asynchronous, service will retry before discarding the event. Therefore, you need to gracefully handle errors.
* *Performance*: Design the function with a warm start.
* *Security*: Design your function with zero trust security with one role per function.

**Azure Functions**

Azure Functions helps you develop and deploy serverless applications. It’s a command-line interface (CLI) that offers structure, automation, and best practices for the deployment of both code and infrastructure, allowing you to focus on building business code that is event-driven. Azure Functions supports multiple programming languages such as JavaScript, Python, C#, F#, Java, etc., with extensive integration options. Durable functions provide stateful capabilities and bindings for Azure Event Hub, and Azure Event Grid helps you to build event-driven architecture.

Azure provides an online editor that is built on Visual Studio Online and provides a well-defined deployment pipeline. You can set up a continuous build and deployment using source code management software such as Team Service, GitHub, or Bitbucket.

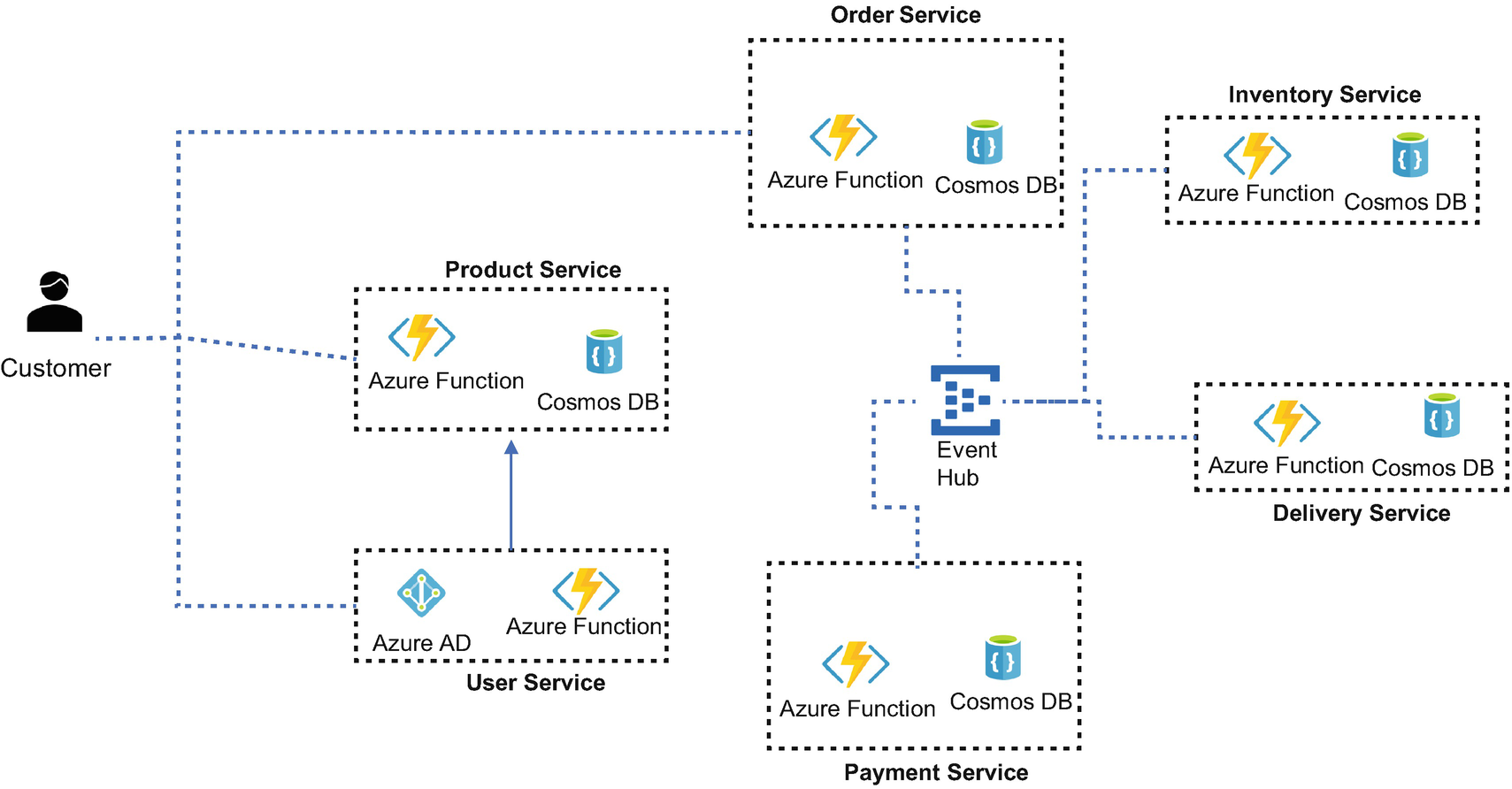
Azure functions are logically grouped into an application container or environment called an *app service*. All the Azure functions within an app service share the same resources such as compute and memory. This enables the deployment as a whole application instead of individual functions.

The API gateway functionalities are built natively into Azure Functions, so they do not require any separate API gateway configuration.

Azure Functions supports several types of event triggers. Cron jobs enable timer-based events for scheduled tasks; for example, OneDrive or SharePoint can be configured to trigger operations in a function.

**Reference Architecture**

The following section provides a few reference architectures for Azure Functions; these reference architectures are just an example to show how functions can be used in your systems. I will use the same ecommerce services as illustrated in the Lambda example and will explain them from the Azure Functions perspective. Figure [7-7](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_7_Chapter.xhtml#Fig7) shows only the logical application architecture, not the deployment architecture, as Azure Functions is combined with Azure App Services for deployment.



***Figure 7-7***

Reference serverless architecture for Azure Functions

* *API gateway*: There is no separate API gateway; the HTTP triggers are built in to Azure Functions.
* *Event hub*: For service-to-service asynchronous communication, you can use any event broker such as Kafka, etc.
* *Azure AD*: For managing and authenticating users, this provides the JSON web tokens used by web services.
* *Azure Functions*: The service implementation communicate with APIs or events.
* *Cosmos DB*: For storage service, you can use any service like MongoDB, Couchbase, Cassandra, etc.

**Best Practices of Azure Functions**

The following are the best practices of using Azure Functions:

* *Avoid long-running functions*: Large, long-running functions can cause unexpected timeout issues.
* *Cross-function communication*: Use Azure Durable Functions and Azure Logic Apps. These were built to manage state transitions and communicate between multiple functions.
* *Stateless*: Functions should be stateless and idempotent wherever possible. Associate any required state information with your data.
* *Write defensive functions*: Design your function so it has the ability to continue from a previous fail point during the next execution.
* *Organize functions for performance and scaling*: Optimize Azure Functions within the function app.
* *Organize functions by privileges*: Apply zero trust security and minimize the number of functions with access to specific credentials.
* Reuse connections to external resources wherever possible.

**Google Cloud Functions**

Google Cloud Functions is an FaaS offering from Google’s serverless paradigm. It is a serverless execution environment for building and connecting cloud services. It is based on an open stack. A cloud function can be written using the JavaScript, Python, Go, or Java runtime, as of writing this book, but its programming language support is expanding.

Google Cloud Functions provides a connective layer of logic that lets you write code to connect and extend any cloud service. It augments existing cloud services and allows you to address various use cases with arbitrary programming logic.

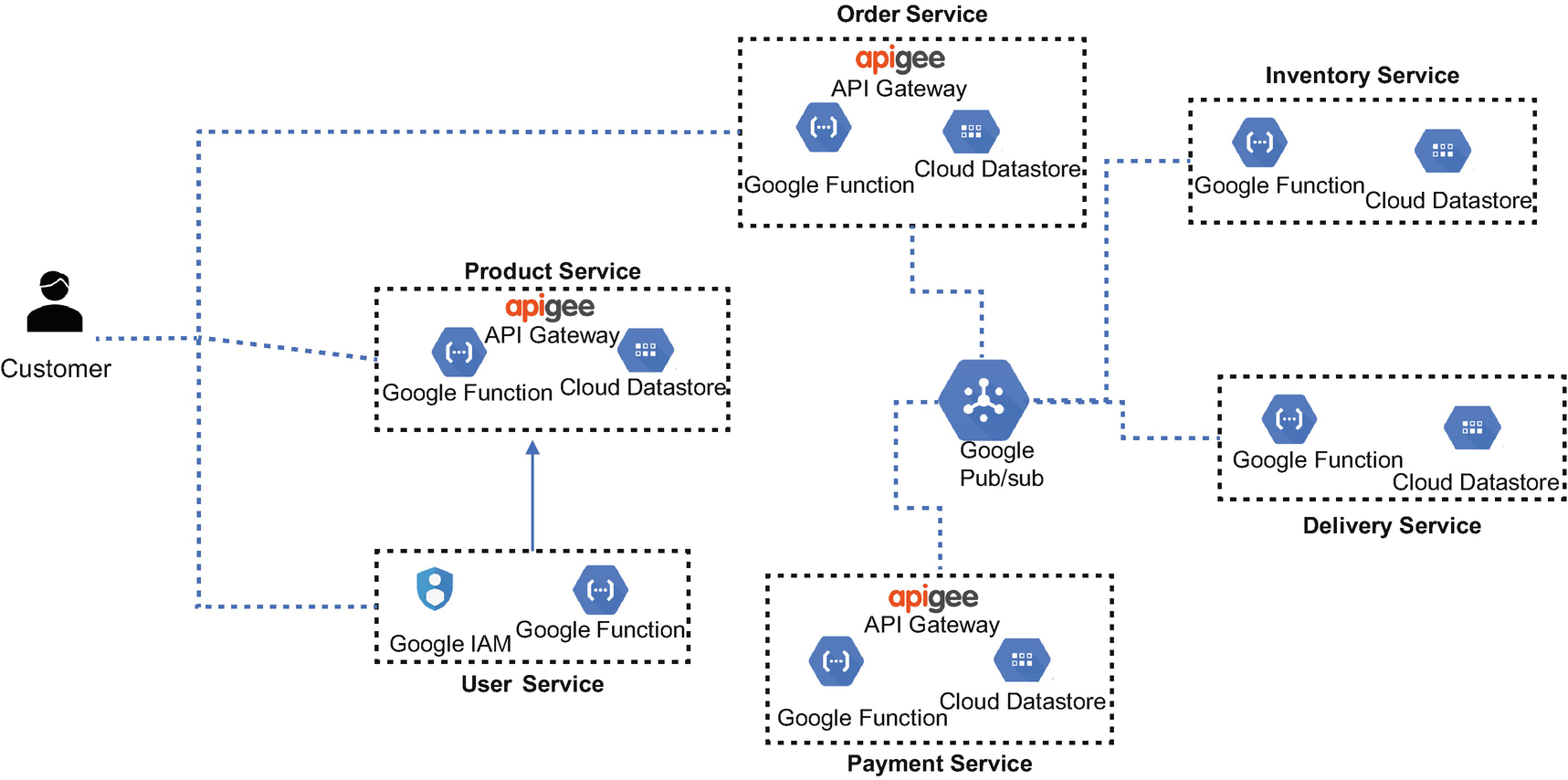
From the security account perspective, it has access to the Google Service Account credentials and is thus seamlessly authenticated with the majority of Google Cloud services.

Google Cloud Functions has a good runtime environment that scales up well and cools down slowly and also has strong observability with the stack driver integration results in exceptional sets of telemetry. Data, files, and stream processing are done in real time using the Google platform. The other services in the Google Cloud help Google enhance time efficiency and simplify technology management and administration.

Google Cloud Functions is still catching up with its peers in terms of programming language support and efficiency.

**Reference Architecture**

This section provides a few Google Cloud Functions reference architecture examples, as shown in Figure [7-8](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_7_Chapter.xhtml#Fig8). This reference architecture is just an example to provide how Google Cloud Functions can be used in your system. I will use the same ecommerce services as illustrated in Lambda and Azure Functions and will explain them from the Google Cloud Functions perspective. Here I am showing only the logical application architecture, not the deployment architecture.



***Figure 7-8***

Reference serverless architecture for Google Cloud Functions

* *API gateway*: For service-to-service synchronous communication and access from mobile and web applications, you can use any other Apigee and also support third-party gateways such as Layer 7, Akana, etc.
* *Google pub/sub*: For service-to-service asynchronous communication, you can use any other event broker such as Kafka, etc.
* *Google IA*: For managing and authenticating users, this provides the JSON web tokens used by web services.
* *Google function*: The service implementation communicates with APIs and events.
* *Cloud datastore*: For storage service, you can use any service such as MongoDB, Couchbase, Cassandra, etc.

**Best Practices of Google Function**

This section describes general best practices for designing and implementing functions via Google Cloud Functions:

* *Write idempotent functions*: Your function should produce the same result every time it is called.
* *Do not start background activities*: Background activity is anything that happens after your function has terminated a function invocation finishes once the function returns or otherwise signals completion, such as calling a callback argument in Node.js background functions. Any code run after graceful termination cannot access the CPU and will not make any progress.
* *Always delete temporary files*: Have local disk storage in the temporary directory as an in-memory file system. Failing to explicitly delete these files results in an out-of-memory error.
* *Use dependencies wisely*: A function is stateless; the execution environment is often initialized from scratch.
* *Use global variables to reuse objects in future invocation*: If you declare a variable in global scope, its value can be reused in subsequent invocations without having to be recomputed.

**FaaS Platform Evaluation Criteria**

FaaS is a key element of cloud native architecture, and each cloud provider has an FaaS platform that is readily available to you, so everything may not be useful to you. Use the Table [7-1](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_7_Chapter.xhtml#Tab1) criteria to evolve the FaaS platforms for your use and rate each framework against these criteria. One thing you need to remember is that not all the frameworks are 100 percent correct, and some frameworks score higher than others. Ask questions like these: What kind of use cases do you want. What are all the other architecture components. Where do you want to deploy them? One thing I would like to get straight is that the serverless technology is still maturing, and not all use cases can be serverless.

***Table 7-1***

FaaS Platform Evaluation Criteria

| Step | Criteria | Details |
| --- | --- | --- |
| **1** | Developer experience | How easy is it to create and deploy new functions? Is there any sample template available to develop a function and list of native development and deployment tools available? |
| **2** | Programming model | What does the programming model support? What is the state of the workloads (for example, stateless, long-running, etc.)? |
| **3** | Runtime execution environment | How do frameworks support building and deployment, and how do they manage cold and warm starts? What are the configuration options, and how easy is it to configure with business code? |
| **4** | Observability | What observability options does the FaaS framework provide (for example, logs, metrics, etc.), and how are machine learning models integrated for predictive analysis? |
| **5** | Integration | What are the various options that frameworks provide for integration? What kind of events can it subscribe to and publish? Is there support for event meshes and service meshes? How are APIs integrated? |
| **6** | Security features | How is access management done? How does it manage serverless specific threats such as denial-of-wallet attacks? |
| **7** | Extensibility | Can it support multicloud vendor configuration? |
| **8** | Testability | How is the framework support for nonfunctional testing such as performance, scale, etc.? |
| **9** | Roadmap | What is the release and upgrade strategy of frameworks? |
| **10** | AI and ML support | How does the framework support artificial intelligence and machine learning models? |

**Backend as a Service or Mobile Backend as a Service**

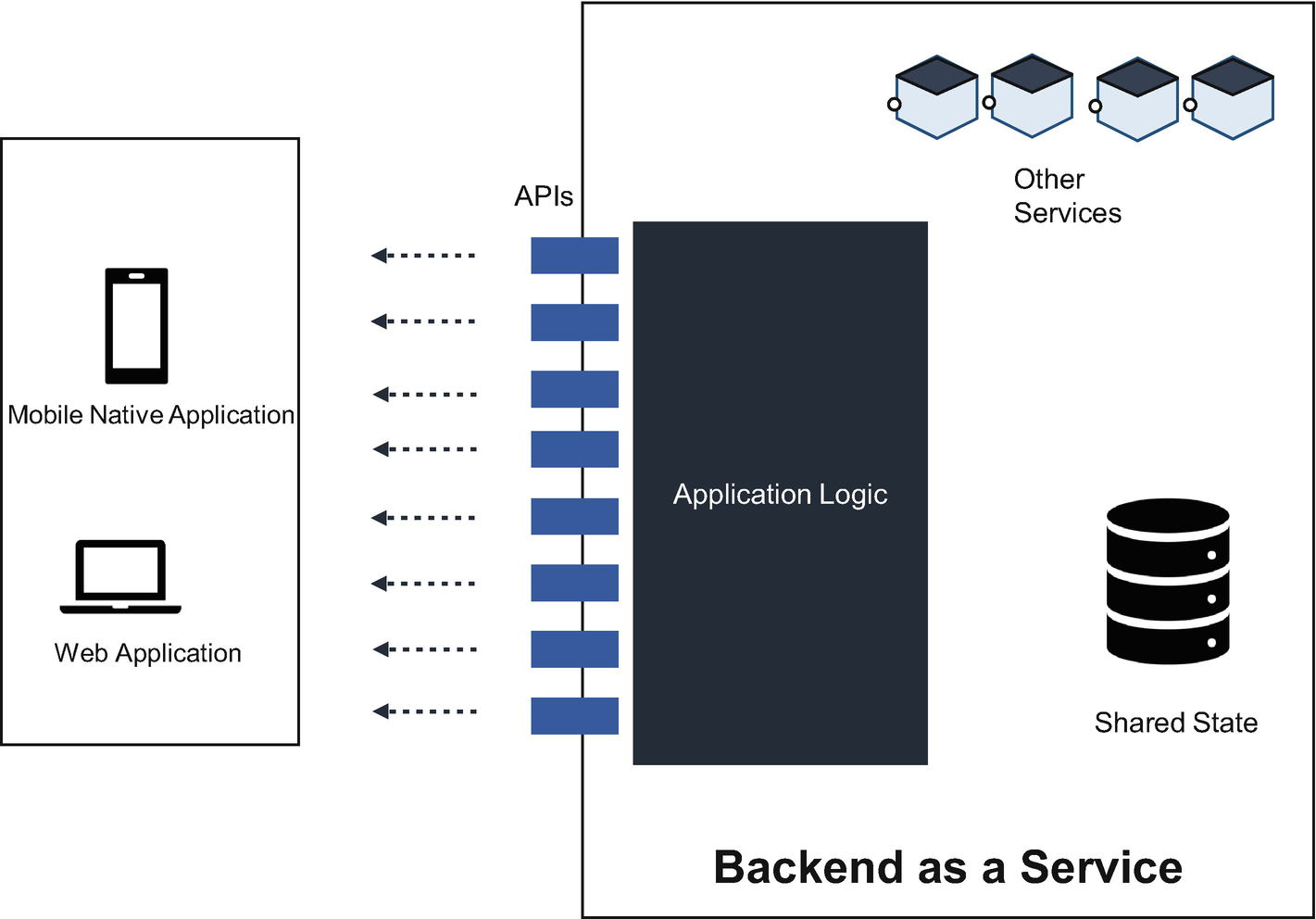
Backend as a service (BaaS) is a cloud computing service model that serves as the middleware that provides developers with ways to connect their web and mobile applications to a cloud-based service. BaaS creates a unified application programming interface (API) and software development kit (SDK) to connect mobile apps to back-end services like cloud storage. This includes key features such as push notifications, social networking integration, location service, and user management, etc., which reduces the development cost and time, as a team does not have to write their code for the various functions. BaaS offers to use existing services.

The BaaS architecture consists of infrastructure as a service (IaaS), platform as a service (PaaS), APIs, and SDKs.

In the traditional architecture or microservices architecture, what steps do you take during mobile or web application development? Usually, you develop a back-end microservice process that contains business logic, authentication, authorization, data storage, integration, etc. There are a lot more other services involved, for example, user analytics, content management, push notification, etc.

BaaS is gaining popularity among enterprises; it is a new model for application development and can lower development costs, allowing the developer to focus on the development process by using APIs or SDKs. It accelerates mobile development and transforms back-end capabilities into services.

The back-end architecture in the cloud empowers the front-end architecture, i.e., mobile and web applications. It comprises hardware and storage located on the cloud service. The cloud service provider manages all the back-end services on behalf of enterprises and acts as a serverless platform. In the Figure [7-9](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_7_Chapter.xhtml#Fig9) architecture, the other services of BaaS are push notification, user management, search functionality, visual development, and file management.



***Figure 7-9***

BaaS architecture service

BaaS eliminates the need for engineers to construct their back-end services and provides the customization features outfitted with common and necessary back-end features.

Is BaaS an advanced version of PaaS? The answer is yes and no. PaaS provides a platform through the cloud for engineers to build their applications. Like BaaS, the PaaS eliminates the need for the developer to build and manage the application back end, but PaaS does not include prebuilt server-side application logic, such as push notifications or user authentication, etc. PaaS offers you more flexibility, while BaaS offers more flexibility and functionality.

Various companies provide BaaS offerings; some are open source, and some are proprietary. The open source BaaS options are Parse, Back4App, Kuzzle, Couchbase, Deployed, etc. The proprietary ones are Pubnub, Appcelerator, PlayFab, Firebase, Kinvey, etc. All these services are hosted on any of the cloud providers or can be downloaded and hosted on any cloud.

**Pros and Cons of BaaS**

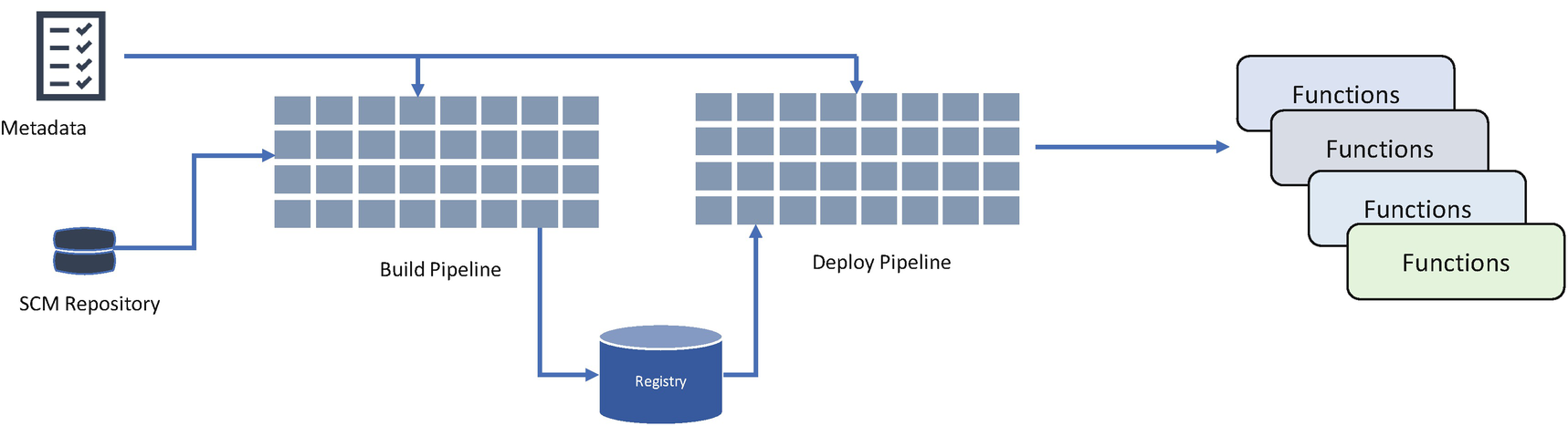
BaaS solves the complexity of cross-platform development and makes it easier to learn the skills needed to create effective back-end processing. You save time on technical use cases such as authentication and authorization, search, data storage, etc., and you don’t need to manage the servers. Also, there is virtually no back-end servers to manage. The whole BaaS platform improves the time to market and reduces the cost of testing, management, etc.

The downside of BaaS is related to vendor lock-in and lower coding flexibility, but most BaaS providers offer an easy way to migrate to others, but in reality, it is not the same. BaaS provides full back-end fixed functionality; therefore, you are losing flexibility to move code from the front-end logic to the back-end logic. From the security front, the platform is isolated and multitenant, so there is a challenge on sensitivity, compliance, regulations, etc.

**Function Deployment**

When your function is deployed in a serverless system, you need to use a deployment pipeline, as shown in Figure [7-10](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_7_Chapter.xhtml#Fig10). While the steps in the pipeline can vary depending on the cloud provider, a few fundamental steps are common to all.

You need to upload the function definition, which contains metadata about the function as well as the implementation code. The metadata includes a unique identifier, name, description, version identifier, runtime language, resource requirements, execution timeout, created date/time, and last modified date/time. When a function is invoked, it is invoked for a specific function version. The version identifier is used to select an appropriate function instance.



***Figure 7-10***

Function deployment pipeline

Along with the metadata, the code and dependency must be provided. Once the metadata is uploaded to the cloud provider, the build process uses it for compilation to produce an artifact. The resultant artifact may be a binary file or a container image.

The starting of an instance function can be a cold or warm start. With a warm start, one or more function instances have already been deployed and are ready to be executed when needed. A cold start takes longer since the function starts for an undeployed state. The function must be deployed and then executed when it is needed.

**When to Use Serverless**

As I explained earlier, the serverless work is based on a single function and is ephemeral and stateless. In terms of computing resources, serverless functions have provided good support in terms of memory and available duration. The duration and memory vary from each cloud vendor, and your function should work within the upper limits of memory and duration.

There are various concerns that serverless architecture gives cloud providers complete control over your services. There are many advantages and use cases that make this a good decision that can benefit the overall outcome of a solution. The following are the best use cases for going serverless:

* **Data Transformation**
* In the data transformation function, you take input, transform it, and provide different output. In this process, no state is required to execute, and the execution is simply based on few rules. The only thing you need to manage is that the data comes in different sizes. If you predictably chunk the data, then you can use a serverless framework.
* **Asynchronous Processing Use Cases**
* As mentioned, the FaaS is event-driven processing. In real terms for an application, it enables the background tasks to be well-connected and remain in the background. For example, the mobile application request is not affected by running in the background and still interacts with an application. You can use it in flight searches, hotel searches, etc.
* **Multimedia Processing Use Cases**
* The function can be created for multimedia processing such as image processing, file upload to the storage, etc.
* **Parallel Compute**
* In parallel computing, you require multiple instances to spin up and execute in parallel to meet the requirements. Using a serverless framework, you can configure with a concurrency model, which essentially translates as one function handling at any one time to scale thousands to meet the parallel processing.
* **Notifications**
* You can create notification functions to work with container-based microservices. The notification can be an email notification or SMS notification, or it can upload any image into the storage notification.
* **Building Restful APIs**
* Serverless frameworks provide a seamless way in which to create a scalable endpoint that processes data in real time. It has the ability to scale and fluctuate as demand changes without the need to maintain the servers.

**Advantages of Serverless Architecture**

Using a serverless architecture provides many important benefits, which explains why it is becoming increasingly popular.

**Reduced Operational Cost**

With a serverless architecture, your code is executed only when it is needed, so you are charged only for the actual compute resources that are used. Organizations moving to a serverless architecture will reduce hardware costs, as they no longer have servers and network infrastructure to support and do not have to hire staff to support all of that infrastructure.

The following are the reductions in the operational costs of serverless systems:

* Savings on the hardware
* Infrastructure management
* Managing the development and deployment of software

**Optimized Resource Utilization**

Serverless empowers you to design an application that scales up and down as per the demands. Serverless will take care of the optimum utilization of resources by managing the technical concerns effectively. This enables saving costs and reducing the impact on the environment.

**Faster Time to Market**

You need to concentrate only on solving business problems by writing good software code. Serverless abstracts away the infrastructure, deployment, and plumbing and wiring activity in an environment. The time to market is greatly reduced and avoids lots of unnecessary issues.

**Ability to Focus on User Experience**

Abstraction from servers allows companies to dedicate more time and resources to developing and improving customer-facing features.

**Fits with Microservices**

The serverless platform paradigm helps design a microservices architecture either by deploying a service in the serverless platform or by collaborating with the container-based microservices by deploying reusable functions as an FaaS.

**The Drawbacks of Serverless Architecture**

Although there are many benefits to using a serverless architecture, there is a potential downside to using serverless, and each vendor comes with their own set of drawbacks that need to be considered by architects creating such systems.

**Standardization**

There is no standardization across various vendors. Each vendor’s serverless implementation has a different approach. For example, API gateway’s features are part of Azure Functions, but AWS Lambda is an external feature. This requires locking into a vendor. If you want to move, then you need to rewrite your code.

**Operations Management**

It is more challenging to debug or do other operationalize activity. Implementation is a black box to you, and you will be unable to find any internal details of serverless. Understanding, anticipating, and predicting these operations is challenging. There are operation limits typically enforced on the duration of execution, size of the function, network utilization, storage capacity, memory usage, thread count, request and response size, etc. The vendor documentation does not have clear details on the operationalization of serverless.

**Tooling Support**

As I mentioned, serverless is still evolving, as compared to the containers. The containers have a lot of support tooling from the various industry players and also from open source other than the cloud vendors. Testing and deployment tools are limited, and the industry may come up with various tools as serverless adoption evolves.

**Security**

As the implementation of serverless is not transparent and is entirely managed by the cloud vendors, you need to rely on the vendor’s security management. You do not have visibility into compliance, regulations, etc.

**Long-Term Tasks**

Serverless platforms are ephemeral functions that execute in a time-boxed manner. Each function must have a well-defined execution boundary. So, the ideal use cases to run as FaaS are deterministic computations that return execution results in a finite amount of time. You need to be careful when architecting your solution for long-running, probabilistic jobs. Running such jobs can incur more costs, which defeats the purpose of adopting serverless.

**Future of Serverless**

In recent years, serverless computing has gained a lot of traction, which has had a large influence on the computing industry. Serverless vendors are constantly innovating and enriching their offerings with better tools such as deployment, testing, and monitoring tools.

In the future, complex technical solutions will move from container to serverless computing and will be implemented and fully managed. The cloud services will provide capabilities in serverless computing platforms in the form of APIs and events.

As I explained in earlier chapters, the adoption of cloud native services has increased, especially after the pandemic. Cloud native services are mainly connected with other services through events. In the future, all services on the cloud and on-premises ecosystem are connected through events. All events related to the enterprise and extended enterprise business can be processed in a serverless manner, regardless of where the event occurs.

VMs and containers are two virtualization technologies with different orientations with strong security and high overheads. Serverless computing requires the highest security and minimum resource overheads and compatibility at the same time. For example, the serverless platform must be able to support arbitrary binary files. This makes it impractical for users to build serverless computing with language-specific VMs. Hence, new lightweight virtualization technologies, such as AWS Firecracker and Google gVisor, have emerged. For example, AWS Firecracker provides a minimal required device model and optimizes kernel loading, enabling startups within 100 milliseconds and minimal memory overhead. As the scale and influence are constantly expanding, it becomes important to implement end-to-end optimization at the framework level, language, and hardware levels based on the load characteristics of serverless computing.

**Summary**

Serverless approaches are designed to handle idle servers that affect enterprises’ balance sheets without offering value; they also remove the cost of building and operating a fleet of servers.

Various cloud vendors offer serverless solutions for long-standing problems by eliminating the servers, containers, disks, and other infrastructure. Serverless is the easiest and fastest way to architecture a reactive, event-based system with a cloud native architecture.

In this chapter, you learned about the design principles, patterns, and use cases of serverless.