**14**

**Microservice Architectures in AWS**

This chapter builds upon the concepts and AWS services you learned about earlier in the areas of serverless architecture and containers in *Chapter 6*, *Harnessing the Power of Cloud Computing,* under the *Learning serverless compute with AWS Lambda and Fargate* section, and in *Chapter 13*, *Containers in AWS*. Now that you know the basics of containers and have discovered what some of the most popular container implementations are today, you can continue to learn about higher-level concepts that use containers to create modern, modular, and nimble applications. If your company operates in the cloud, it is very likely that it is taking advantage of the capabilities of the cloud. One architectural pattern that is extremely popular nowadays is microservice architecture.

In this chapter, you will do a deep dive into the ins and outs of microservice patterns. Specifically, you will learn about the following topics:

* Understanding microservices
* Microservice architecture patterns
* Building layered architecture
* Benefits of **event-driven architecture** (**EDA**)
* Disadvantages of EDA
* Reviewing microservices best practices
* Implementing **Domain-Driven Design** (**DDD**)

By the end of this chapter, you will have learned about microservice architecture design and popular pattern-like architectures, such as event-driven architecture and Domain-Driven Design, and their pros and cons. Let’s get started.

**Understanding microservices**

Like many ideas that become popular in technology, it is hard to pin down an exact definition of microservices. Different groups co-opt the term and provide their own unique twist on the definition, but the popularity of microservices is hard to ignore. It might be the most common pattern used in new software development today. However, the definition has not stayed static and has evolved over time.

Given these caveats, let’s try to define what a microservice is.

A **microservice** is a software application that follows an architectural style that structures the application as a service that is loosely coupled, easily deployable, testable, and organized in a well-defined business domain. A **loosely coupled system** is one where components have little or no knowledge about other components and there are few or no dependencies between these components.

In addition, a certain consensus has been reached, to some degree, around the concept of microservices. Some of the defining features that are commonly associated with microservices are the following:

* In the context of a **microservice architecture**, services communicate with each other over a network with the purpose of accomplishing a goal using a technology-agnostic protocol (most often HTTP).
* Services can be deployed independently of each other. The deployment of a new version of one of the services, in theory, should not impact any of the associated services.
* Services are assembled and built around business domains and capabilities.
* Services should be able to be developed using different operating systems, programming languages, data stores, and hardware infrastructure and still be able to communicate with each other because of their common protocol and agreed-upon **APIs** (**Application Programming Interfaces**).
* Services are modular, small, message-based, context-bound, independently assembled and deployed, and decentralized.
* Services are built and released using an automated process – most often a **continuous integration** (**CI**) and **continuous delivery** (**CD**) methodology.
* Services have a well-defined interface and operations. Both consumers and producers of services know exactly what the interfaces are.
* Service interfaces normally stay the same or at least have background compatibility when code is changed. Therefore, clients of these services do not need to make changes when the code in the service is changed.
* Services are maintainable and testable. Often, these tests can be fully automated via a CI/CD process.
* Services allow for the fast, continuous, and reliable delivery and deployment of large and complex projects. They also facilitate organizations to evolve their technology stack.

Microservices are the answer to monolithic architectures that were common in mainframe development. Applications that follow a monolithic architecture are notoriously hard to maintain, tightly coupled, and difficult to understand. Also, microservices aren’t simply a layer in a modularized application like was common in early web applications that leveraged the **Model/View/Controller** (**MVC**) pattern. Instead, they are self-contained, fully independent components with business functionality and clearly delineated interfaces. This doesn’t mean that a microservice might not leverage other architectural patterns and have its own individual internal components.

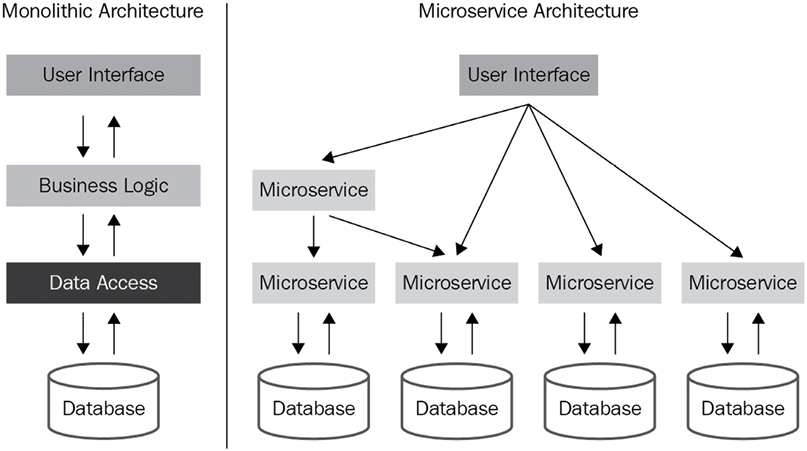
Doug McIlroy is credited with describing the philosophy surrounding Unix; microservices implement the Unix philosophy of *“Do one thing and do it well*.*”*

Martin Fowler describes microservices as those services that possess the following features:

* Software that can leverage a CI/CD development process. A small modification in one part of the application does not require the wholesale rebuild and deployment of the system; it only requires rebuilding, deploying, and distributing a small number of components or services.
* Software that follows certain development principles such as fine-grained interfaces.

Microservices fit hand in glove with the cloud and serverless computing. They are ideally suited to be deployed using container and serverless technology. In a monolithic deployment, if you need to scale up to handle more traffic, you will have to scale the full application. When using a microservice architecture, only the services that are receiving additional calls need to be scaled. Depending on how the services are deployed and assuming they are deployed in an effective manner, you will only need to scale up or scale out the services that are taking additional traffic and you can leave the servers that have the services that are not in demand untouched.

Microservices have grown in popularity in recent years as organizations are becoming nimbler. In parallel, more and more organizations have moved to adopt a DevOps and CI/CD culture. Microservices are well suited for this. Microservice architectures, in some ways, are the answer to monolithic applications. A high-level comparison between the two is shown in the following diagram:



*Figure 14.1: Monolithic versus microservice architectures*

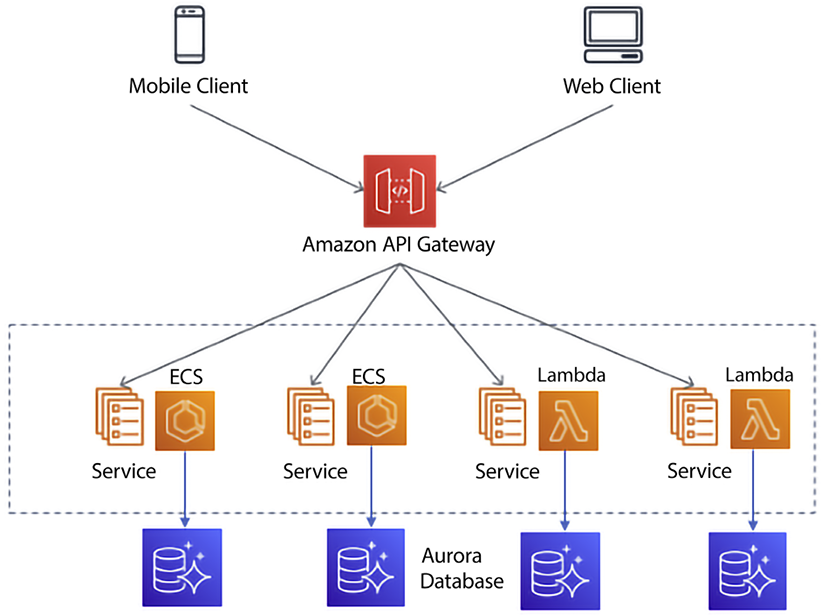
In a monolithic architecture, communication occurs across the whole application independent of business boundaries. Initially, and for simple applications, this architecture may be appropriate, but complexity increases quickly as the number of business domains that the application handles increases.

In a microservice architecture, each service is typically responsible for a specific business capability and is built and deployed independently of other services. As part of this separation, each service typically has its own database and API. Having a separate database for each service allows for better scalability, as the data storage and retrieval needs of each service can be optimized independently. It also allows for more flexibility in terms of the technology choices for each service, as different services can use different types of databases depending on their needs.

Having a separate API for each service allows for a clear separation of concerns and promotes loose coupling between services. It also allows for easier evolution of the services, as changes to one service’s API will not affect other services.

This can also facilitate the versioning of the service. However, it’s important to note that there are also some trade-offs with this approach. One of the main trade-offs is the increased complexity of managing and integrating multiple services.

The following diagram shows a microservice architecture in AWS at a high level. In this architecture a request goes through API Gateway and is routed to a different microservice based on the container manager Amazon ECS and the serverless microservices built on AWS Lambda. All microservices have their own Amazon Aurora database instances.



*Figure 14.2: Microservice architectures in AWS*

With this architecture, requests to the microservices would go through API Gateway, which would then route the requests to the appropriate microservice running in a container on ECS. The microservice would then interact with its own Aurora database to retrieve or store data:

* **API Gateway**: This service can be used to create and manage APIs for microservices. It allows for the creation of RESTful and WebSocket APIs, as well as the ability to handle authentication and authorization, traffic management, and caching.
* **ECS**: This service can be used to deploy and manage the containers that run the microservices. ECS allows you to easily scale and update the services, and it also provides service discovery and load balancing.
* **Aurora**: This service can be used as a managed relational database for microservices. Aurora can be used to create, configure, and manage relational databases that can be used by the microservices; it also provides automatic backups, software patching, and replication for high availability. You can also use Aurora Serverless to further reduce operational overhead.

This is just one example of how to build a microservice architecture using AWS services, and there are many other services and configurations that can be used depending on the specific requirements of the application.

There are other components that go into microservice architecture, like security, networking, caching, etc., which are not mentioned in the diagram to keep it simple. However, you will learn about these in more detail in upcoming sections. In a microservice architecture, the boundaries between services are well defined according to business domains. This enables applications to scale more smoothly and increases maintainability.

In this section, you saw the fundamentals of microservices. Next, we will learn about two popular architecture patterns that are often used when creating microservices.

These patterns are as follows:

* Layered architecture
* Event-driven architecture

In the following sections, you will learn about these common architectures in detail and go through the advantages and disadvantages of each of them.

**Layered architecture**

This pattern is quite common in software development. As indicated by the name, in this pattern, the code is implemented in layers. Having this layering enables the implementation of *“separation of concerns*.” This is a fancy way of saying that each layer focuses on doing a few things well and nothing else, which makes it easier to understand, develop, and maintain the software.

The topmost layer communicates with users or other systems. The middle layer handles the business logic and routing of requests, and the bottom layer’s responsibility is to ensure that data is permanently stored, usually in a database.

Having this separation of concerns or individual duties for each layer allows us to focus on the most important properties for each layer. For example, in the presentation layer, accessibility and usability are going to be important considerations, whereas, in the persistence layer, data integrity, performance, and privacy may be more important. Some factors will be important regardless of the layer. An example of a ubiquitous concern is security. But, by having these concerns separate, it enables teams to not require personnel that are experts in too many technologies. With this pattern, we can hire UI experts for the presentation layer and database administrators for the persistence layer. It also provides a clear delineation of responsibilities. If something breaks, it can often be isolated to a layer, and once it is, you can reach out to the owner of the layer.

From a security standpoint, a layered architecture offers certain advantages over more monolithic architectures. In a layered architecture, you normally only place the presentation layer services in a public subnet and place the rest of the layers in a private subnet. This ensures that only the presentation layer is exposed to the internet, minimizing the attack surface. As a best practice, you should only put the load balancer in the public domain with the protection of a web application firewall.

If a hacker wanted to use the database in an unauthorized manner, they would have to find a way to penetrate through the presentation layer and the business logic layer to access the persistence layer. This by no means implies that your system is impenetrable. You still want to use all security best practices and maybe even hire a white hat group to attempt to penetrate your system. An example of an attack that could still happen in this architecture is a SQL injection attack. That said, the layered architecture will limit the attack surface to the presentation layer only, so this architecture is still more secure than a monolithic architecture.

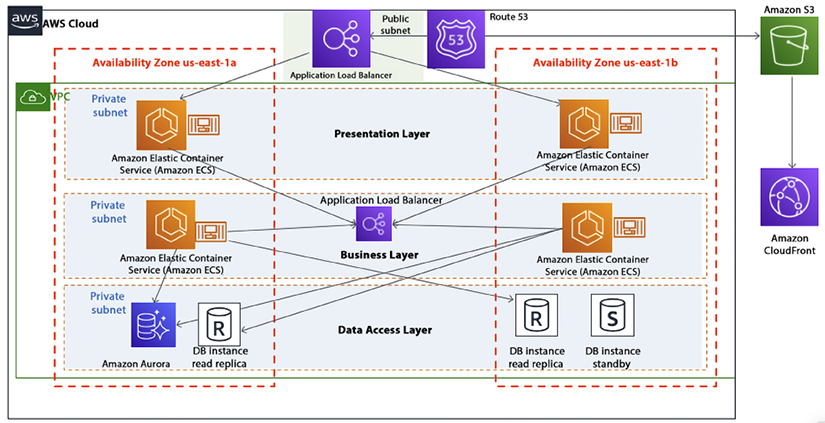
Another advantage of having a layered architecture is gaining the ability to swap out a layer without having to make modifications to any of the other layers. For example, you may decide that AngularJS is no longer a good option for the presentation layer and instead you want to start using React. Or you may want to start using Amazon Aurora PostgreSQL instead of Oracle. If your layers were truly independent and decoupled, you would be able to convert the layers to the new technology without having to make modifications to the other layers.

In a microservice-based architecture, the layers are typically broken down as follows:

* **Presentation layer**: This layer is responsible for handling the user interface and presenting the data to the user. This layer can be further divided into the client-side and server-side parts.
* **Business layer**: This layer is responsible for implementing the business logic of the system. It communicates with the presentation layer to receive requests and with the data access layer to retrieve and update data.
* **Data access layer**: This layer is responsible for communicating with the database and other data storage systems. It provides an abstraction layer between the business layer and the data storage, allowing the business layer to focus on the logic and not worry about the details of data access.

You can keep an additional layer that hosts services for security, logging, monitoring, and service discovery. Each layer is a separate microservice, which can be developed, deployed, and scaled independently. This promotes the flexibility, scalability, and maintainability of the system. In microservice architectures, it is also possible to add more layers in between as per the requirements, such as a security layer, API Gateway layer, service discovery layer, etc.

The following diagram shows a three-layer architecture in AWS, where the user experience frontend is deployed in the presentation layer, all business logic is handled by the business layer, and data is stored in the data access layer:



*Figure 14.3: Three-layer microservice architecture in AWS*

As shown in the preceding diagram, all web and application servers are deployed in containers managed by Amazon ECS, where requests are routed through an elastic load balancer and the entire environment is protected by a VPC. You can also choose Lambda or Fargate to have a completely serverless implementation, along with Aurora Serverless for your database.

Just because you are using a layered approach, it does not mean that your application will be bug-free or easy to maintain. It is not uncommon to create interdependencies among the layers. When something goes wrong in a layered architecture, the first step in troubleshooting the issue is to identify which layer the problem is occurring in. Each layer has a specific role and responsibility, and the problem will typically be related to that layer’s functionality. Once the layer has been identified, you can focus on the specific components within that layer that are causing the problem.

Here are some examples of how troubleshooting might proceed for different layers:

* **Presentation layer**: If the problem is related to the user interface, you might investigate issues with the client-side code, such as JavaScript errors or browser compatibility issues. On the server side, you might investigate issues with the routing, rendering, or handling of user input.
* **Business layer**: If the problem is related to the business logic, you might investigate issues with the implementation of the logic itself, such as incorrect calculations or validation rules. You might also investigate issues with communication between the business layer and other layers, such as the presentation layer or data access layer.
* **Data access layer**: If the problem is related to data access, you might investigate issues with the database connections, queries, or transactions. You might also investigate issues with the mapping between the data model and the database schema, such as incorrect column names or data types.

Finally, if the problem is related to the underlying infrastructure, you might investigate issues with the network connections, security configurations, or service discovery mechanisms. Once you have identified the specific component causing the problem, you can use various debugging and monitoring tools to gather more information and diagnose the issue.

It’s worth noting that having a clear and well-defined layered architecture can make troubleshooting more straightforward and efficient, as it allows you to focus on a specific layer and its related components instead of having to consider the entire system as a whole.

**Event-driven architecture (EDA)**

**Event-driven architecture** (**EDA**) is another pattern commonly used when implementing microservices. When the event-driven pattern is used, creating, messaging, processing, and storing events are critical functions of the service. Contrast this with the layered pattern we just looked at, which is more of a request/response model and where the user interface takes a more prominent role in the service.

Another difference is that layered architecture applications are normally synchronous whereas an EDA relies on the asynchronous nature of queues and events.

More and more applications are being designed using EDA from the ground up. Applications using EDA can be developed using a variety of development stacks and languages. EDA is a programming philosophy, not a technology and language. EDA facilitates code decoupling, making applications more robust and flexible. At the center of EDA is the concept of events. Let’s spend some time understanding what they are.

**Understanding events**

To better understand the event-driven pattern, let’s first define what an event is. Events are messages or notifications that are generated by one component of the system and consumed by other components. These events represent something significant or important that has occurred within the system and that other components need to know about in order to take appropriate action. Essentially, an event is a change in state in a system. Examples of changes that could be events are the following:

* A modification of a database
* A runtime error in an application
* A request submitted by a user
* An EC2 instance fails
* A threshold is exceeded
* A code change that has been checked into a CI/CD pipeline
* A new customer is registered in the system
* A payment is processed
* A stock price changes
* A sensor reports a temperature reading
* A user interacts with a mobile app

Not all changes or actions within a system are considered events in an EDA. For example, a change to a configuration setting or a log message might not be considered an event because it does not represent something significant that other components of the system need to know about.

It’s also worth noting that the distinction between an event and a non-event can be context-dependent and may vary depending on the specific implementation of the EDA.

In some cases, certain changes or actions that would not typically be considered events might be treated as such if they are deemed important or relevant to certain components or use cases within the system.

In the next section, we’ll discuss two other critical elements in EDA: the concepts of producers and consumers.

**Producers and consumers**

Events by themselves are useless. If a tree falls in the forest and no one is around to hear it or see it fall, did it really fall? The same question is appropriate for events. Events are worthless if someone is not consuming them, and in order to have events, producers of the events are needed as well. These two actors are essential components of EDA. Let’s explore them at a deeper level:

* **Producers**: An event producer first detects a change of state and if it’s an important change that is being monitored, it generates an event and sends a message out to notify others of the change.
* **Consumers**: Once an event has been detected, the message is transmitted to a queue. Importantly, once the event has been placed in the queue and the producer forgets about the message, consumers fetch messages from the queue in an asynchronous manner. Once a consumer fetches a message, they may or may not perform an action based on that message. Examples of these actions are as follows:
  + Triggering an alarm
  + Sending out an email
  + Updating a database record
  + Opening a door
  + Performing a calculation

In essence, almost any process can be a consumer action.

As you can imagine, due to the asynchronous nature of EDA, it is highly scalable and efficient.

EDA is a loosely coupled architecture. Producers of events are not aware of who is going to consume their output and consumers of events are not aware of who generated the events. Let’s now learn about two popular types of models designed around EDA.

**EDA models**

There are a couple of ways to design an event-driven model. One of the main design decisions that needs to be made is whether events need to be processed by only one consumer or by multiple consumers. The first instance is known as the event streamingpattern. The second pattern is most commonly known as the publish and subscribe pattern. EDA can be implemented using either of these two main patterns. Depending on the use case, one pattern may be a better fit than the other. Let’s learn more about these two models.

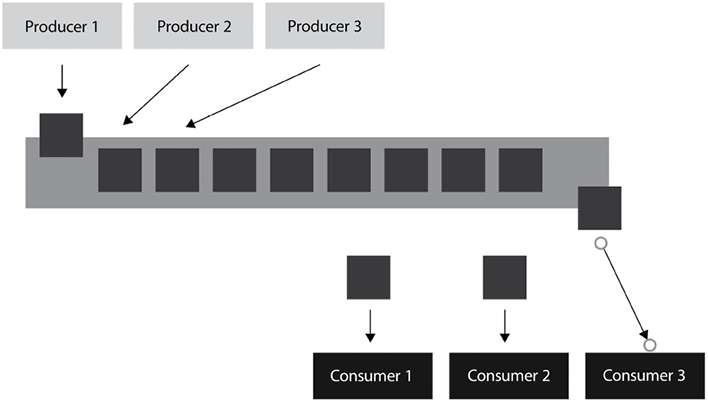
**Event streaming (a message queuing model)**

In the event streaming model, events are *“popped off”* the queue as soon as one of the consumers processes the message. In this model, the queue receives a message from the producer and the system ensures that the message is processed by one and only one consumer.

Event streaming is well suited for workloads that need to be highly scalable and can be highly variable. Adding capacity is simply a matter of adding more consumers to the queue, and we can reduce capacity just as easily by removing some of the consumers (and reducing our bill). In this architecture, it is extremely important that messages are processed by only one consumer. In order to achieve this, as soon as a message is allotted to a consumer, it is removed from the queue. The only time that it will be placed back in the queue is if the consumer of the message fails to process the message and it needs to be reprocessed.

Use cases that are well suited for this model are those that require that each message is processed only once but the order in which the messages are processed is not necessarily important.

Let’s look at a diagram of how an event streaming architecture would be implemented:



*Figure 14.4: Event streaming model*

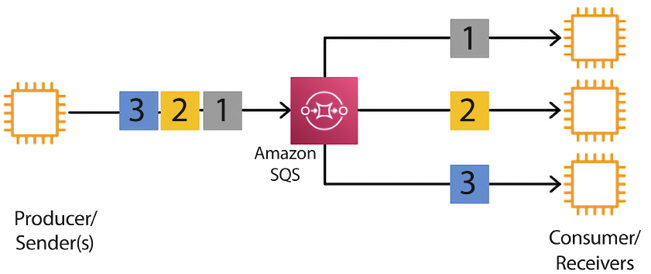
In the preceding diagram, we have multiple producers generating events and placing them into a single queue (on the left-hand side). We also have multiple consumers consuming events off the queue (on the right-hand side). Once a consumer takes an event from the queue, it gets removed and no other consumer will be able to consume it. The only exception is if there is an error and the consumer is unable to complete the consumption of the event. In this case, we should put some logic in our process to put the unconsumed event back in the queue so that another consumer can process the event.

Let’s make this more concrete with a real-life example.

**Example scenario**

In order to visualize this model, think of the queues that are common in banks, where you have a single queue that feeds into all the tellers. When a teller becomes available, the first person in the queue goes to that teller for processing and so forth. The customer needs to visit only one teller to handle their transaction. As tellers go on a break or new tellers come in to handle the increased demand, the model can gracefully and transparently handle these changes. In this case, the bank customers are the producers – they are generating events (for example, making a check deposit), and the bank tellers are the consumers – they are processing the events that the customers are creating.

You can use Amazon **Simple Queue Service** (**SQS**) to implement a queue model as shown in the following diagram.



*Figure 14.5: Event-driven message queuing model in AWS*

As shown in the preceding diagram, here, messages are coming from the producer and going into SQS. Amazon SQS is a serverless, scalable queue service. It allows consumers to take messages from the queue and process them as per their needs. If your application is using an industry-standard queue service like JMS or RabbitMQ, you can use Amazon MQ, which provides managed support for RabbitMQ and Apache ActiveMQ.

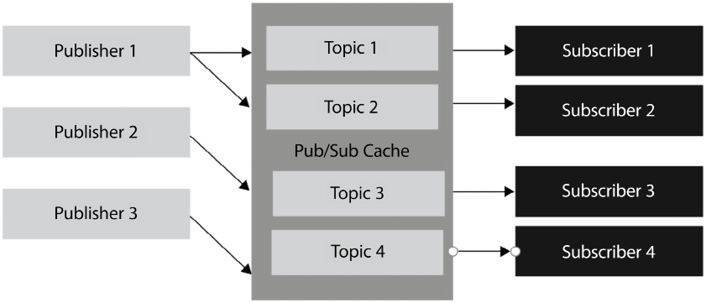
Now let’s move on and learn about another type of event-driven model – the pub/sub model.

**Publish and subscribe model (pub/sub model)**

As happens with event streaming, the **publish and subscribe** model (also known as the **pub/sub** model) assists in communicating events from producers to consumers. However, unlike event streaming, this model allows several consumers to process the same message. Furthermore, the pub/sub model may guarantee the order in which the messages are received.

As the publishing part of the name indicates, message producers broadcast messages to anyone that is interested in them. You express interest in the message by subscribing to a topic.

The pub/sub messaging model is suited for use cases in which more than one consumer needs to receive messages. In this model, many publishers push events into a **pub/sub cache** (or queue). The events can be classified by topic. As shown in the following diagram, subscribers listen to the queue and check for events being placed in it. Whenever events make it to the queue, the consumers notice them and process them accordingly. Unlike the model in the previous section, when a subscriber sees a new event in the queue, it does not pop it off the queue; it leaves it there and other subscribers can also consume it, and perhaps take a completely different action for the same event.



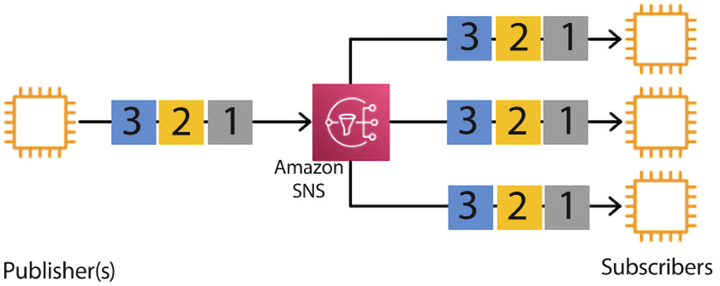
*Figure 14.6: The pub/sub model*

Optionally, the events in the cache can be classified by topic, and subscribers can subscribe only to the topics they are interested in and ignore the rest. The following diagram shows the pub/sub model as achieved by Amazon **Simple Notification Service** (**SNS**).

A managed service provided by AWS for a pub/sub model is EventBridge. AWS EventBridge is a serverless event bus service that allows you to connect different applications and services together using a pub/sub model. With EventBridge, you can create rules that automatically trigger specific actions in response to events from various sources, such as changes in an S3 bucket or the creation of a new item in a DynamoDB table. This allows you to easily integrate different parts of your application and automate tasks without having to write custom code. EventBridge supports both custom events and events from AWS services, making it a powerful tool for building event-driven architectures.

**Example scenario**

An example of this is a stock price service. In this case, typically, many market participants are interested in receiving prices in real time on a topic of their choosing (in this case, the topics are the individual tickers). In this case, the order in which the order tickers are received is incredibly important. If two traders put in a purchase to buy stock for the same price, it is critical that the system processes the order that was received first. If it doesn’t, the market maker might get in trouble with accusations of front-running trades.



*Figure 14.7: Event-driven pub/sub model in AWS*

As shown in the preceding diagram, all messages from different publishers go to SNS, where multiple consumers are subscribed to receive them. SNS fans out messages to all the subscribers for further processing, as per the application’s requirements.

The pub/sub model is frequently used with stateful applications. In a stateful application, the order in which the messages are received is important, as the order can impact the application state.

**Benefits of EDA**

EDA can assist an organization to obtain an edge over its competitors. This edge stems from the benefits that the pub/sub model can provide. Some of the benefits are explained in the following sub-sections.

**No more polling**

The pub/sub model delivers the benefit of real-time events through a *“push”* delivery mechanism. It eliminates the need to constantly be fetching sources to see whether data has changed. If you use a polling mechanism, you will either waste resources by checking for changes when no changes have occurred, or you will delay actions if changes occur when you haven’t polled.

Using a push mechanism minimizes the latency of message delivery. Depending on your application, delays in message delivery could translate into a loss of millions of dollars.

**Example**: Let’s say you have a trading application. You want to buy stock only when a certain price is reached. If you were using polling, you would have to constantly ping every so often to see if the price had changed. This has two problems:

* Computing resources will have to be used with every ping. This is wasteful.
* If the price changes in between pings, and then changes again, the trade may not execute even though the target price was reached.

With events, the ping will be generated only once when the target price is reached, greatly increasing the likelihood that the trade will happen.

**Dynamic targeting**

EDA simplifies the discovery of services and does so in an effortless and natural way, minimizing the number of potential errors. In EDA, there is no need to keep track of data consumers and, instead, interested parties simply subscribe to the topics that are of interest. If there are parties interested in the messages, the messages get consumed by all of them. In the pub/sub model, if there aren’t any interested consumers, the message simply gets broadcast without anyone taking any action.

**Example**: Continuing with our trading application example, let’s assume that each stock represents a topic. Letting users of the application select what topic/stock interests them will greatly reduce the number of events generated and therefore will reduce resource consumption.

**Communication simplicity**

EDA minimizes code complexity by eliminating direct point-to-point communication between producers and consumers. The number of connections is greatly reduced by having a central queue where producers place their messages and consumers collect messages.

**Example**: Let’s assume that our trading application has 10 stock shares and 10 users. If we didn’t have an intermediate queue to hold the events, every stock share would have to be connected to every user, for a total of 100 connections. But having a queue in the middle would mean that we only have 10 connections from the stock to the queue and 10 connections from the users to the queue, giving us a total of 20 connections, greatly simplifying the system.

**Decoupling and scalability**

The pub/sub model increases software flexibility. There is no explicit coupling between publishers and subscribers. They all are decoupled and work independently of each other. Having this decoupling promotes the individual development of services, which, in turn, allows us to deploy and scale these services independently. Functionality changes in one part of the application should not affect the rest of the application so long as design patterns are followed, and the code is truly modularized. So long as the agreed-upon APIs stay stable, making a change in the publisher code should not affect the consumer code.

**Example**: In our trading application, if a new stock ticker is added, users don’t need a new connection to the new stock. We simply create a connection from the new stock to the queue and now anybody can listen for events in that new topic. Something similar happens when new users get added. The user just needs to specify which stocks they are interested in. Nothing else needs to be changed in the system. This makes the overall architecture quite scalable.

As we have seen, the advantages of EDA are many. However, no software solution is perfect, and EDA is no exception. Let’s now investigate the disadvantages that come with it.

**Disadvantages of EDA**

As with other technologies, EDA also has drawbacks. Some of the drawbacks are explained in the following sub-sections.

**EDA is not a silver bullet**

It is worth noting that, like any other technology, the EDA pattern should not be viewed as a solution that can solve all problems. A problem may not require the added complexity of setting up a message queue. We might only require a “point-to-point” communication channel because we don’t foresee having additional producers or consumers. The EDA pattern is quite popular with new IoT applications, but it is not suitable for other use cases. If your application is synchronous in nature and it only requires accessing and updating a database, using EDA may not be necessary and might be overcomplicated. It is important to determine how much interactivity and inter-process communication will be required in our application before recommending EDA as a pattern for a given problem. EDA applications require some effort to maintain and troubleshoot when problems arise (by having to check consumers, producers, and queues) and an individual problem might not warrant their use.

**Example**: What if our trading application only focused on one stock share? In that particular example, we might want to avoid the complexity of creating queues, topics, and so on and keep it simple without using a queue.

**When things go wrong**

Like any other technology that depends on an underlying infrastructure, it is possible in an EDA implementation for messages to get lost for various reasons, including the failure of hardware components. Dealing with such failures can be difficult to troubleshoot and even more difficult to find a solution to recover from. These issues stem from the asynchronous nature of the architecture. This property makes the resulting applications massively scalable but with the downside of potentially losing messages. Overcoming this shortcoming can be challenging.

**Example**: Due to the asynchronous nature of EDA applications, it is not easy to troubleshoot them. In our trading application, we might lose a message due to hard failure. We obviously want to minimize or even eliminate these occurrences. However, trying to replicate the behavior to debug them may be difficult, if not impossible. You can use managed, serverless, AWS-native services such as EventBridge to reduce the risk.

**Microservices best practices**

As with any technology, the devil is in the details. It is certainly possible to create bad microservices. Let’s delve into how some common pitfalls can be avoided and some recommended best practices.

**Best practice #1 – decide whether microservices are the right tool**

The world’s leading technology companies, such as eBay, Facebook, Amazon, Microsoft, Twitter, and PayPal, are all heavy users of microservice architecture and rely on it for much of their development. However, it’s not a panacea. As technologists, once we get a hammer, everything looks like a nail. Make sure that your particular use case is best suited for this architecture. If it’s hard to break down your application into functional domains, a microservice architecture might not be the best choice.

**Best practice #2 – clearly define the requirements and design of the microservice**

Creating microservices, like other software projects, requires preparation and focus. A sure way for a software project to fail is to start coding without having a clear goal in mind for the function of the software.

Requirements should be written down in detail and approved by all stakeholders. Once the requirements are completed, a design should be created using a language and artifacts that are understood by all parties involved, including domain experts.

A clear distinction should be made between business requirements and functions, the services that will be provided, and the microservices that will be implemented to provide the services. Without this delineation, it is likely the microservices will be too big and not fragmented enough and no benefit will be delivered from using a microservice architecture. On the other hand, it is also possible for your design to have too many microservices and for you to over-engineer the solution. If there are too many microservices, the solution will be difficult to maintain, understand, and troubleshoot.

**Best practice #3 – leverage Domain-Driven Design to create microservices**

Later in this chapter, we will learn about the Domain-Driven Design methodology. We will learn more about it in a moment, but Domain-Driven Design is ideally suited for the development of microservices. Domain-Driven Design is a set of design principles that allow us to define an object-oriented model using concepts and nomenclature that all stakeholders can understand using a unified model language. It allows all participants in the software definition process to fully understand the relevant business domains and deliver better microservices because you can get buy-in and understanding from everyone more quickly.

**Best practice #4 – ensure buy-in from all stakeholders**

Software development involves many parties in an organization: developers, architects, testers, domain experts, managers, and decision-makers, among others. In order to ensure the success of your project, you need to make sure to get buy-ins from all of them. It is highly recommended that you get approval from all stakeholders at every major milestone – particularly during the business requirement and design phase. In today’s Agile culture, the initial requirements and design can often change, and in those instances, it is also important to keep stakeholders updated and in agreement.

Deploying a microservice entails much more than just technology. Getting approval and mindshare from the status quo is key. This cultural transformation can be arduous and expensive. Depending on the team’s exposure to this new paradigm, it might take a significant effort, especially if they are accustomed to building their applications in a monolithic manner.

Once you start delivering results and business value, it might be possible to start getting into a cadence and a harmonious state with all team members. And for that reason, it is important to make sure that you start delivering value as soon as possible. A common approach to achieve this is to deliver a minimum viable product that delivers the core functionality in order to start deriving value, and to then continue building and enhancing the service once the minimum viable product is deployed to production and starts being used.

**Best practice #5 – leverage logging and tracing tools**

One of the disadvantages of using a microservice architecture is the added burden of logging and tracing many components. In a monolithic application, there is one software component to monitor. In a microservice architecture, each microservice generates its own logging and error messages. With a microservice architecture, software development is simplified but operations become a little more complicated. For this reason, it is important that your services leverage the logging and tracing services that AWS offers, such as AWS CloudWatch, AWS X-Ray, and AWS CloudTrail, where the logging and error messages generated are as uniform as possible. Ideally, all the microservice teams will agree on the logging libraries and standards to increase uniformity. Two products that are quite popular to implement logging are the **ELK** stack (consisting of **Elasticsearch, Logstash, and Kibana**) and Splunk.

**Best practice #6 – think microservices first**

Software development can be a fine art more than a hard science. There are always conflicting forces at play. You want to deliver functionality in production as quickly as possible but, at the same time, you want to ensure that your solution endures for many years and is easily maintainable and easily expandable. For this reason, some developers like using a monolithic architecture at the beginning of projects and then try to convert it to a microservice architecture.

If possible, it is best to fight this temptation. The tight coupling that will exist because of the architecture choice will be difficult to untangle once it is embedded. Additionally, once your application is in production, expectations rise because any changes you make need to be thoroughly tested. You want to make sure that any new changes don’t break existing functionality. You might think that code refactoring to ease maintenance is a perfectly valid reason to change code in production. However, explaining to your boss why the production code broke when you were introducing a change that did not add any new functionality will not be an easy conversation. You may be able to deliver the initial functionality faster using a monolithic architecture, but it will be cumbersome to later convert it into a more modular architecture.

It is recommended to spend some time on correctly designing your microservices’ boundaries from the start. If you are using an Agile methodology, there will no doubt be some refactoring of microservices as your architecture evolves and that’s okay. But do your best to properly design your boundaries at the beginning.

**Best practice #7 – minimize the number of languages and technologies**

One of the advantages of the microservice architecture is the ability to create different services using different technology stacks. For example, you could create Service A using Java, the Spring MVC framework, and MariaDB and you could create Service B using Python with a Postgres backend. This is doable because when Service A communicates with Service B, they will communicate through the HTTP protocol and via the RESTful API, without either one caring about the details of the other’s implementation.

Now, just because you can do something, doesn’t mean you should do it. It still behooves you to minimize the number of languages used to create microservices. Having a small number of languages, or maybe even using just one, will enable you to swap people from one group to another, act more nimbly, and be more flexible.

There is a case to be made that one stack might be superior to the other and more suited to implement a particular service, but any time you have to deviate from your company’s standard stack you should make sure that you have a compelling business case to deviate from the standards and increase your technological footprint.

**Best practice #8 – leverage RESTful APIs**

A key feature of the microservice pattern is to deliver its functionality via a RESTful API. **RESTful APIs** are powerful for various reasons, among them the fact that no client code needs to be deployed in order to start using them, as well as the fact that they can be self-documenting if implemented properly.

**Best practice #9 – implement microservice communication asynchronously**

Whenever possible, communication between microservices should be asynchronous. One of the tricky parts about designing microservices is deciding the boundaries between the services. Do you offer granular microservices or do you only offer a few services? If you offer many services that perform a few tasks well, there will undoubtedly be more inter-service communication.

In order to perform a task, it may be necessary for Service A to call Service B, which, in turn, needs to call Service C. If the services are called synchronously, this interdependency can make the application brittle. For example, what happens if Service C is down? Service A won’t work and will hopefully return an error. The alternative is for the services to communicate asynchronously. In this case, if Service C is down, Service A will put a request in a queue and Service C will handle the request when it comes back online. Implementing asynchronous communication between services creates more overhead and is more difficult than synchronous communication, but the upfront development cost will be offset by increasing the reliability and scalability of the final solution.

There are many ways to implement asynchronous communication between microservices. Some of them are as follows:

* **Amazon SNS**: SNS is a distributed pub/sub service. Messages are pushed to any subscribers when messages are received from the publishers.
* **Amazon SQS**: SQS is a distributed queuing system. With SQS, messages are NOT pushed to the receivers. Receivers pull messages from the queue and once they pull a message, no one else can receive that message. The receiver processes it, and the message is removed from the queue.
* **Amazon Kinesis**: Amazon SNS and Amazon SQS are good options with simple use cases. Amazon Kinesis is more appropriate for real-time use cases that need to process terabytes of data per minute.
* **Amazon Managed Streaming for Kafka** (**MSK**): Amazon MSK is a fully managed service for Apache Kafka. As well as the rest of the options listed below, it is an open source solution. With Amazon MSK, you can create and manage Kafka clusters in minutes, without the need to provision or manage servers, storage, or networking. The service automatically handles tasks such as software patching, backups, and monitoring, enabling you to focus on your application logic. Apache Kafka supports both a queue and a pub/sub architecture.
* **Amazon MQ**: Amazon MQ is a managed message broker service for Apache ActiveMQ, which is another popular open source messaging tool. With Amazon MQ, you can set up a message broker using ActiveMQ in minutes, without the need to provision or manage the underlying infrastructure. The service automatically handles tasks such as software patching, backups, and monitoring, enabling you to focus on your application logic. Amazon MQ supports multiple messaging protocols, including JMS, NMS, AMQP, STOMP, MQTT, and WebSocket. This allows you to easily connect to your existing applications and devices and use the messaging protocol that best suits your needs.
* **AWS EventBridge**: EventBridge allows you to publish events to a centralized event bus, which can then be consumed by other microservices in a decoupled manner. This decoupling allows microservices to communicate with each other without requiring a direct connection or knowledge of each other’s implementation details.

**Best practice #10 – implement a clear separation between microservice frontends and backends**

Even today, many backend developers have an outdated perspective about what it takes to develop UIs and tend to oversimplify the complexities involved in constructing user-friendly frontends. The UI can often be neglected in design sessions. A microservice architecture with fine-grained backend services that has a monolithic frontend can run into trouble in the long run. There are great options out there that can help create sharp-looking frontends. Some of the most popular frontend web development frameworks currently are the following:

* Vue
* React
* Angular

However, picking the hottest SPA tool to develop your frontend is not enough. Having a clear separation between backend and frontend development is imperative. The interaction and dependencies between the two should be absolutely minimal, if they are not completely independent.

As new UIs become more popular or easier to use, we should be able to swap out the frontend with minimal interruptions and changes to the backend.

Another reason for having this independence comes about when multiple UIs are required. For example, our application may need a web UI, an Android application, and an Apple iOS application.

**Best practice #11 – organize your team around microservices**

On a related note to the previous best practice, there might be different teams for individual microservices and it’s important to assign ownership of each of these services to individuals in your team. However, hopefully, your team is as cross-functional as possible and team members can jump from one microservice to another if the need arises. In general, there should be a good reason to pull one team member from the development of one service to another, but when this does happen, hopefully, they are able to make the leap and fill the gap.

In addition, the team should have a decent understanding of the overall objectives of the projects, as well as knowledge of the project plan for all services. Having a narrow view of only one service could prove fatal to the success of the business if they don’t fully understand the business impact that a change in their service could have on other services.

**Best practice #12 – provision individual data stores for each individual microservice**

Separating your garbage into recyclables and non-recyclables and then watching the garbage collector co-mingle them can be frustrating. The same is true of microservices that have well-defined and architected boundaries and then share the same database. If you use the same database, you create strong coupling between the microservices, which we want to avoid whenever possible. Having a common database will require constant synchronization between the various microservice developers. Transactions will also get more complicated if there is a common database.

Having a separate data store makes services more modular and more reusable. Having one database per microservice does require that any data that needs to be shared between services needs to be passed along with the RESTful calls, but this drawback is not enough to not separate service databases whenever possible.

Ideally, every microservice will have an individual allocation for its data store. Every microservice should be responsible for its own persistence. Data can be reused across services, but it should only be stored once and shared via APIs across the services. However, whenever possible, you should avoid data sharing across microservices as data sharing leads to service coupling. This coupling negates some of the advantages of the separation of concerns of a microservice architecture, so it should be avoided as much as possible.

**Best practice #13 – self-documentation and full documentation**

A well-designed RESTful API should be intuitive to use if you choose your domain name and operation names correctly.

Take special care to use labels for your APIs that closely match your business domains. If you do this, you won’t need to create endless documents to support your application. However, your documentation should be able to fill the gaps and take over where the intuitiveness of your API ends. One of the most popular tools to create this documentation is a tool called **Swagger**. You can learn more about the Swagger tool here: <https://swagger.io/>.

**Best practice #14 – use a DevOps toolset**

Another methodology that goes hand in hand with microservice development (in addition to Domain-Driven Design) is the popular **DevOps paradigm**. Having a robust DevOps program in place along with a mature CI/CD pipeline will allow you to develop, test, and maintain your microservices quickly and effortlessly.

A popular combination is to use Jenkins for deployment and Docker as a container service with GitHub. AWS CodePipeline can be used to automate an end-to-end DevOps pipeline.

**Best practice #15 – invest in monitoring**

As we learned in the preceding section regarding the disadvantages of microservices, they can be more difficult to monitor and troubleshoot than legacy monolithic architectures. This increased complexity must be accounted for, and new monitoring tools that can be adapted to the new microservice architecture need to be used.

Ideally, the monitoring solution offers a central repository for messages and logs regardless of what component of the architecture generated the event.

The monitoring tools should be able to be used for each microservice, and the monitoring system should facilitate root cause analysis. Fortunately, AWS offers a nice selection of monitoring services, including the following:

* Amazon CloudWatch
* Amazon CloudTrail
* Amazon X-Ray

To learn more about these and other monitoring services in AWS, you can visit:

<https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/monitoring_ec2.html>

**Best practice #16 – two pizzas should be enough to feed your team**

This is a rule popularized by Jeff Bezos. He famously only invites enough people to meetings so that two large pizzas can feed the attendees. Bezos popularized the *two-pizza* rule for meetings and project teams to encourage a decentralized, creative working environment and to keep the start-up spirit alive and well.

This rule’s goal is to avoid groupthink. *Groupthink* is a phenomenon that occurs when you have large groups and people start going with the consensus instead of feeling comfortable pushing back against what they think are bad ideas. In some ways, it is human nature to be more hesitant to disagree in large groups.

It is not uncommon for members of the group that are lower in the corporate hierarchy to be intimidated by authority figures such as their boss and people with bigger titles. By keeping groups small and encouraging dialogue, some of this hesitancy may be overcome and better ideas may be generated.

Bezos’ idea to keep meetings and teams small to foster collaboration and productivity can be backed up by science. During his 50 years of studies and research of teams, J. Richard Hackman concluded that 4 to 6 is the optimal number of team members for many projects and that teams should never be larger than 10.

According to Hackman, communication issues “*grow exponentially as team size increases.*” Perhaps counterintuitively, the larger a team is, the more time will be used to communicate, reducing the time that can be used productively to achieve goals.

In the context of microservice development, the two-pizza rule is also applicable. You don’t want your microservice development and deployment teams to be much bigger than a dozen people or so. If you need more staff, you are probably better off splitting the microservice domains so that you can have two teams creating two microservices rather than one huge team creating an incredibly big and complex microservice.

Obviously, there is no hard rule about exactly how many people is too many people, but at some point, the number becomes too big and unmanageable. For example, having a 100-person monolithic team with no hierarchy or natural division in it most likely would be too unmanageable.

**Best practice #17 – twelve-factor design**

A popular methodology that is out there to enhance microservice development is one dubbed *the twelve-factor design*. This methodology accelerates and simplifies software development by making suggestions such as ensuring that you are using a version control tool to keep track of your code.

The twelve-factor design is a methodology for building **software-as-a-service** (**SaaS**) applications that are easy to scale and maintain. It was first introduced in a 2011 article by Adam Wiggins, co-founder of Heroku, a cloud platform for building and deploying web applications.

The twelve factors are:

1. **Code base**: One code base is tracked in revision control for many deployments
2. **Dependencies**: Explicitly declare and isolate dependencies
3. **Config**: Store config in the environment
4. **Backing services**: Treat backing services as attached resources
5. **Build, release, run**: Strictly separate build and run stages
6. **Processes**: Execute the app as one or more stateless processes
7. **Port binding**: Export services via port binding
8. **Concurrency**: Scale out via the process model
9. **Disposability**: Maximize robustness with fast startup and graceful shutdown
10. **Dev/prod parity**: Keep development, staging, and production as similar as possible
11. **Logs**: Treat logs as event streams
12. **Admin processes**: Run admin/management tasks as one-off processes

By following these principles, the twelve-factor methodology aims to make it easy to scale and maintain SaaS applications by breaking them down into small, loosely coupled services that can be run in different environments and easily deployed on cloud-based platforms. You can learn more about this methodology here: <https://12factor.net/>.

Many of the best practices mentioned in this section don’t just apply to microservice development but are also useful in software development in general. Following these practices from the beginning of your project will greatly increase the chances of a successful implementation that is on time and on budget, as well as making sure that these microservices are useful, adaptable, flexible, and easily maintainable.

In today’s world, software is used to solve many complicated problems. From meeting worldwide demand for your e-commerce site to enabling a real-time stock trading platform, many companies, big and small, are leveraging Domain-Driven Design to bring their products and services to market on time. Let’s take a look into the Domain-Driven Design pattern.

**Domain-Driven Design (DDD)**

**Domain-Driven Design**(**DDD**) might fall into the shiny new object category, as many people see it as the latest trendy pattern. However, DDD builds upon decades of evolutionary software design and engineering wisdom. To better understand it, let’s briefly look at how the ideas behind DDD came about with a brief overview of **Object-Oriented Programming** (**OOP**).

DDD has its roots in the OOP concepts pioneered by Alan Key and Ivan Sutherland. The term OOP was coined by Alan Key around 1966 or 1967 while in grad school. OOP is a powerful programming paradigm that allows for the creation of well-structured, maintainable, and reusable code, and is widely used in the development of modern software applications.

OOP is a way of thinking about programming that is based on the concept of “objects.” Objects can be thought of as instances of a class, and are used to represent and manipulate real-world entities. OOP uses objects and their interactions to design and write programs. It’s a bit like building a house, where you use different blocks (objects) to build different rooms (programs), and you can use the same blocks (objects) in different ways to build different rooms (programs).

Imagine you’re making a video game where you control a character, like Mario in Super Mario Bros. In OOP, you would create an “object” that represents Mario, and give it properties like its position on the screen, how fast it can move, and how many lives it has. You would also give it “methods” that tell it what to do, like moving left or right, jumping, and so on. The whole game would be made up of many different objects, each with its own properties and methods. For example, there would be objects for the Goombas (enemies), pipes, coins, and so on. All these objects would interact with each other in a way that makes sense for the game.

OOP also has some other concepts, like inheritance, polymorphism, encapsulation, and abstraction, which help to make the code more organized and easier to maintain.

OOP languages like Java, C++, Python, C#, etc. are widely used in the industry. These languages provide features like classes, objects, inheritance, polymorphism, encapsulation, and so on to build OOP-based applications.

Ivan Sutherland created an application called Sketchpad, an early inspiration for OOP. Sutherland started working on this application in 1963. In this early version of an OOP application, objects were primitive data structures displayed as images on the screen. They started using the concept of inheritance even in those early days. Sketchpad has some similarities with JavaScript’s prototypal inheritance.

OOP came about because developers and designers were increasingly ambitious in tackling more complex problems, and procedural languages were insufficient. Another seminal development was the creation of a language called **Simula**. Simula is considered the first fully OOP language. It was developed by two Norwegian computer scientists—Ole-Johan Dahl and Kristen Nygaard.

A lot of development and many projects relied heavily on OOP for a long time. Building upon the advances of OOP, Eric Evans wrote the book *Domain-Driven Design: Tackling Complexity in the Heart of Software* in 2003. In his book, Evans introduced us to DDD and posited that DDD represents a new, better, and more mature way to develop software building on the evolution of **Object-Oriented Analysis and Design** (**OOAD**).

DDD builds upon OOP by providing a set of principles and practices for designing software that models complex business domains. While OOP focuses on the implementation of objects and their interactions, DDD focuses on the modeling of the business domain and the creation of a rich, domain-specific language that accurately captures its complexities. DDD uses this language to drive the design and implementation of the software. DDD and OOP have a lot of similarities, and many of the concepts of OOP are present in DDD. Some of the key OOP concepts that are present in DDD include:

* **Encapsulation**: DDD encourages the use of encapsulation to hide the internal details of domain objects and expose their behavior through a public interface. This allows the domain objects to be treated as black boxes that can be interacted with through a set of predefined methods, without the need to understand their internal workings.
* **Inheritance**: DDD uses inheritance to model “is-a” relationships between domain objects. For example, a specific type of product might inherit from a more general product class.
* **Polymorphism**: DDD uses polymorphism to model “like” relationships between domain objects. For example, different types of products might share some common behavior, but also have unique behaviors specific to their type.
* **Abstraction**: DDD encourages the use of abstraction to break down complex problems into smaller, more manageable parts.

In addition to the above OOP concepts, DDD also introduces several other concepts, such as bounded contexts, aggregates, domain services, value objects, entities, repositories, and so on, which help to model the business domain in a more accurate and efficient way.

**Definition of a domain**

Now that we have taken a drive down memory lane regarding the history of DDD, let’s first nail down what a domain is before we delve into the definition of DDD. According to the Oxford English Dictionary, one of the definitions of a domain is “*a sphere of knowledge or activity*.”

Applying this definition to the software realm, domain refers to the subject or topic area that an application will operate in. In application development terms, the domain is the *sphere of knowledge and activity used during application development* and the activity that is specific to a particular business or application. It includes the concepts, rules, and relationships that make up the problem space that the application is trying to solve. The domain is the core of the application, and it is where the business logic resides.

For example, if you are building an e-commerce application, the domain would include concepts such as products, customers, orders, and payments. It would also include rules and relationships such as how products are categorized, how customers can place orders, and how payments are processed. The domain is the starting point for DDD, and it is the source of the domain-specific language that is used throughout the development process. By creating a rich, domain-specific language, the development team can communicate more effectively and create a more accurate and efficient design. The domain model is the representation of the domain in the software application; it is the core of the application and it contains the business logic, rules, and relationships. The model should be created from the domain experts’ knowledge; it should be accurate and should be able to capture the complexity of the domain.

Another common way this word is used is to refer to the domain layer or the domain logic. Many developers also refer to this as the business layer or the business logic. In software development, the business layer refers to the layer of the application that contains the business logic and implements the business rules of the application. The business logic is the set of rules and processes that govern how an application behaves and interacts with the domain. Business objects are the objects that represent the business entities and implement the business logic.

A business rule is a specific rule or constraint that governs the behavior of the business objects and the application as a whole. These rules can be specific to a business or industry, and they dictate how the business operates and interacts with the outside world.

For example, in an e-commerce application, a business rule might be that customers must be at least 18 years old to make a purchase. This rule would be implemented in the business logic and would be enforced by the application when a customer attempts to make a purchase. The business layer also contains the business object, a representation of a business entity, such as a customer, product, or order. These objects encapsulate the data and behavior of the corresponding business entity, and they provide a way for the application to interact with the domain. The business objects are responsible for implementing the business logic, including enforcing the business rules.

Suppose a bank account holder tries to retrieve a certain amount of money from their bank, and their account does not have enough balance to honor the request. In that case, the bank should not allow the account holder to retrieve any funds (and charge them an *insufficient funds fee*).

Can you spot the potential business objects in this example? Pause for a second before we give you the answer to see if you can figure it out. Two candidates are these:

* Bank account holder
* Bank account

Depending on the application, you might not want to model the holder as a separate object and rely on the account, but it will depend on the operations that need to be performed on the objects. If you decide to merge the account with the holder, this might generate data duplication (for example, you might store the same address twice when a holder has two accounts and only one address). This issue might not be an issue at all in your implementation.

**Principles of DDD**

As we mentioned earlier, Eric Evans coined the term DDD, so who better to ask for a definition than Evans? As it turns out, even for him, the definition has been a moving target, and the definition he initially gave in his book is no longer his preferred definition. Moreover, defining DDD is not a simple exercise, and Evans defines DDD in multiple ways. This is not necessarily a bad thing—by having multiple definitions, we can cover the term using different lenses.

DDD focuses on understanding and modeling the complex business domains that software systems support. The goal of DDD is to align software systems with the underlying business domains they support, resulting in systems that are flexible, scalable, and maintainable. Here are the core principles of DDD in detail:

* **Ubiquitous language**: This principle states that a common language should be established between domain experts and software developers. This language should be used consistently throughout the development process, including modeling, coding, testing, and documentation. This helps to ensure that everyone involved in the development process has a shared understanding of the domain and reduces the risk of misunderstandings and miscommunication. For example, a financial institution’s domain experts and software developers establish a common language to describe financial instruments such as bonds, stocks, and mutual funds. They use this language consistently throughout the development process, which helps to ensure that everyone involved has a shared understanding of these concepts and reduces the risk of misunderstandings and miscommunication.
* **Bounded contexts**: This principle states that different parts of a complex business domain should be divided into separate contexts. Each context should have a clear boundary and its own ubiquitous language. This helps to ensure that each context can be understood, modeled, and developed in isolation, which can reduce complexity and improve maintainability. For example, a retail company’s business domain is divided into separate contexts for customer management, order management, inventory management, and shipping management.
* **Strategic design**: This principle states that the overall architecture and design of the software system should align with the long-term goals and vision of the business domain. This helps to ensure that the system is flexible and scalable, and can support the changing needs of the business over time. For example, a manufacturing company’s software system is designed with a strategic focus on supporting the company’s long-term goals and vision. The system is designed to be flexible and scalable, and to support the changing needs of the company over time, such as changes to the product line or manufacturing processes.

By following the above principles, DDD helps to ensure that software systems are aligned with the core business domains they are intended to support. Let’s look into the components of DDD.

**Components of DDD**

While you have learned about the principles of DDD, to build your architecture, it is important to understand various components of it. The following are the key components that make up DDD:

* **Context mapping**: Context mapping is a technique for defining and organizing the different contexts within a business domain. Each context represents a distinct part of the business and has its own set of business rules and processes. The goal of context mapping is to identify the boundaries of each context and to define a common language that is used within each context. For example, a retail company may have a context for its online store, a context for its brick-and-mortar store, and a context for its fulfillment center. Each of these contexts would have a different set of business rules and processes, and a common language that is used within each context would help to ensure clear communication between the different parts of the business.
* **Domain model**:The domain model is the heart of DDD and represents the business domain in the form of entities, value objects, services, and aggregates. The domain model provides a way to understand and describe the business domain and to align the software system with the business domain.

For example, in a retail company, the domain model might include entities for products, customers, and orders, and value objects for prices and addresses.

* **Entity**: An entity is an object that represents a core business concept and has an identity that distinguishes it from other objects. An entity should encapsulate business logic and represent a meaningful, persistent part of the business domain. For example, in a retail company, the customer entity might include information about the customer, such as their name, address, and order history, as well as methods for performing actions, such as placing an order or updating their information.
* **Value object**: A value object is an object that represents a value or attribute and lacks an identity of its own. A value object should be freely used and shared, and should not change once created. For example, in a retail company, the price value object might represent the cost of a product, while the address value object might represent the shipping address for an order.
* **Aggregate**:An aggregate is a cluster of objects that should be treated as a single unit with regard to data changes. An aggregate should have a root object and a clear boundary that defines what objects belong to the aggregate. The goal of aggregates is to ensure consistency and maintain data integrity when making changes to the data. For example, in a retail company, the order aggregate might include the order entity, the customer entity, and the products that are part of the order.
* **Service**: A service is an object that encapsulates business logic that doesn’t fit neatly into entities or value objects. A service should represent a business capability or process. For example, in a retail company, the order service might handle the process of creating and submitting an order, including checking inventory levels and calculating shipping costs.
* **Repository**: A repository is a pattern that defines a data access layer that abstracts the persistence of entities and aggregates. The goal of a repository is to provide a way to retrieve and store data in persistent storage, such as a database, while abstracting the underlying storage mechanism. For example, the order repository might provide a way to retrieve an order by its order number or to store a new order in the database.
* **Factory**: A factory is a pattern that defines a way to create objects, typically entities or aggregates, in a consistent and maintainable way. The goal of a factory is to provide a way to create objects in a standardized way, with any necessary dependencies, without having to write repetitive code. For example, the order factory might provide a way to create a new order object, complete with the customer and product entities that are part of the order.
* **Modules**: Modules are a way of organizing the code and defining boundaries within a system. A module is a collection of related entities, value objects, services, and repositories that work together to provide a specific business capability. Modules are used to separate the concerns of the system and to reduce complexity. Modules should be autonomous and self-contained and maintain their own integrity and consistency. An example of a module could be a module for handling customer orders in an e-commerce system. This module could include entities such as Order, OrderItem, Customer, and Payment, as well as services such as OrderService, PaymentService, and ShippingService.

Now you have learned about the principles and components of DDD, let’s understand how you can implement it in the AWS platform.

**Implementing DDD in AWS**

Every application you build is associated with solving a specific business problem, especially when solving real-life problems belonging to an industry domain. An industry use case can be very complex, as seen in the previous section, where we used a retail industry use case to help understand various components of DDD. Now let’s understand how to design this complex architecture using services provided by AWS. Implementing an Amazon.com-like e-commerce application using DDD in AWS would be a complex and multi-step process, but here is a general outline of the steps involved:

1. **Define the business domain**: Start by defining the different contexts within the business domain, such as the shopping context, the product context, and the fulfillment context. Create a domain model that represents the core business concepts, such as products, customers, and orders, and define their relationships with each other.
2. **Choose the right AWS services**: Based on the domain model, choose the right AWS services to implement the different components of DDD. For example, Amazon DynamoDB can be used to store entities and aggregate roots, Amazon SQS can be used to implement services, and Amazon S3 can be used for file storage.
3. **Implement the entities and value objects**: Implement the entities and value objects as DynamoDB tables, each with its own set of attributes and methods. Ensure that the data model is consistent with the domain model and that the entities and value objects are properly encapsulated.
4. **Implement the aggregates**: Implement the aggregates as a set of DynamoDB tables, with the root aggregate as the main table. Ensure that the aggregate boundaries are well defined and that the data is consistent within the aggregate.

Implement the aggregates as classes that represent a set of related entities and that enforce consistency within the aggregate.

1. **Implement the services**: Implement the services as classes using AWS Lambda functions, which use SQS to handle messages. Ensure that the services are aligned with the business processes and that they have access to the necessary data and entities.
2. **Implement the repositories**: Implement the repositories as DynamoDB Data Access Objects, which provide a way to store and retrieve data in persistent storage. Ensure that the repositories are aligned with the domain model and that they provide a consistent way to access data.
3. **Implement the factories**: Implement the factories as AWS Lambda functions, which use DynamoDB tables to store and retrieve data. Ensure that the factories are aligned with the domain model and that they provide a consistent way to create objects.
4. **Implement the user interface**: Implement the user interface as a web application that is hosted in AWS and that communicates with the backend services and repositories. Ensure that the user interface is aligned with the domain model and that it provides a consistent and intuitive experience for customers.
5. **Deploy and test the system**: Deploy the system in AWS using the appropriate AWS services and tools, such as AWS CloudFormation or AWS Elastic Beanstalk. Test the system to ensure that it is working as expected and that the data is consistent and accurate.
6. **Monitor and optimize**: Monitor the system using AWS CloudWatch and other AWS tools to ensure that it is running efficiently and that any issues are quickly addressed. Optimize the system as needed to improve performance and reduce costs.

You can choose an OOP language such as Java or Python to implement the entities, value objects, aggregates, services, repositories, and factories. In this case, objects and classes are used to represent the core business concepts and their attributes and behaviors. Here is an example of how objects and classes might be used in an e-commerce website:

* **Product**: A product is a core business concept in an e-commerce website and represents a physical or digital item that can be sold. A product object might have attributes such as name, description, price, and image. A product class might have methods such as addToCart(), removeFromCart(), and getDetails().
* **Customer**: A customer is another core business concept in an e-commerce website and represents a person who can purchase products. A customer object might have attributes such as name, address, email, and phone number. A customer class might have methods such as signUp(), login(), and updateProfile().
* **Order**: An order is a key business concept in an e-commerce website and represents a request by a customer to purchase one or more products. An order object might have attributes such as order number, customer, products, and total cost. An order class might have methods such as placeOrder(), cancelOrder(), and getOrderHistory().
* **Cart**: A cart is a useful business concept in an e-commerce website and represents temporary storage for products that a customer has selected for purchase. A cart object might have attributes such as products and total cost. A cart class might have methods such as addProduct(), removeProduct(), and checkout().

These objects and classes can be used in combination to implement the business processes and workflows of an e-commerce website, such as adding products to a cart, placing an order, and updating the customer’s profile. By using OOP techniques, such as inheritance, encapsulation, and polymorphism, it is possible to build a scalable, maintainable, and reusable system that meets the needs of the business.

The above steps are just a general outline and the implementation details will vary depending on the specific requirements of your e-commerce business use case. However, by following the principles of DDD and using the right AWS services, you can build a scalable, reliable, and efficient system that meets the needs of your business.

**Reasons to use DDD**

In this section, we will learn about the most compelling benefits of DDD and what makes DDD so powerful. These benefits are described in the following list:

* **Better alignment with business goals**: DDD emphasizes the importance of a deep understanding of the business domain and its needs, leading to better alignment between the software system and the business goals.
* **Increased developer productivity**: By using clear and ubiquitous language and concepts, DDD helps to reduce misunderstandings and improve communication between developers, stakeholders, and domain experts. This leads to faster development and fewer mistakes.
* **Improved maintainability**: DDD encourages the use of a rich and expressive domain model, which can help to make the software system more understandable and maintainable over time.
* **Improved scalability**: DDD promotes the use of modular, scalable, and flexible architecture patterns, which can help to ensure that the software system can be easily adapted to meet changing business needs.
* **Improved domain knowledge**: By working closely with domain experts, developers can gain a deeper understanding of the business domain, which can help to inform future development efforts and lead to better software solutions.
* **Better testing and validation**: DDD encourages the use of **test-driven development** (**TDD**) and **behavior-driven development** (**BDD**) techniques, which can help to ensure that the software system meets the needs of the business and can be easily validated and tested.

The use of DDD can lead to improved software development outcomes and better alignment with business goals, resulting in increased productivity; improved maintainability, scalability, and domain knowledge; and better testing and validation.

**Challenges with DDD**

As is the case with any technology and any design philosophy, DDD is not a magic bullet. Now we will examine the challenges that you might encounter when implementing the DDD methodology in your projects in your organization:

* **Domain expertise**: Your project might have experienced developers and architects who are well versed in the tools being used, but if at least some of your team members don’t have domain expertise in the domain being modeled, then the project is destined to fail. If you don’t have this domain expertise, it’s probably best not to use DDD and perhaps not even start your project until someone on your team acquires this skill set, regardless of the methodology used.
* **Iterative development**: Agile development has become a popular methodology. A well-implemented Agile program allows companies to deliver value quicker and for less cost. DDD heavily relies on iterative practices such as Agile. One of the key benefits of combining DDD and Agile is the ability to quickly and iteratively refine the software system as the needs of the business change over time. This can help to ensure that the software system remains aligned with business goals and is capable of adapting to changing business requirements. However, enterprises often struggle to transition from the traditional and less flexible waterfall models to the new methodologies.
* **Technical projects**: There is no magic hammer in software development, and DDD is no exception. DDD shines with projects that have a great deal of domain complexity. The more complicated the business logic of your application, the more relevant DDD is. DDD is not well suited for applications with low domain complexity but a lot of technical complexity.

An example of low domain complexity with high technical complexity could be a system for tracking weather data. The domain itself is relatively simple, with a limited number of concepts, such as temperature, humidity, and wind speed. However, the technical complexity of the system could be high, due to the need to process large amounts of data in real time, handle multiple data sources, and display the data in a user-friendly way. An example of high domain complexity with low technical complexity could be a banking system. The domain itself is complex, with many concepts, such as accounts, transactions, loans, and investments. However, the technical complexity of the system may be low, as the implementation could be based on established and well-understood banking software systems. DDD is well suited for projects with high domain complexity and low technical complexity, where a deep understanding of the business domain is essential to the success of the project. This is because DDD focuses on building a rich and expressive domain model, which is used as the foundation for the software system, and to ensure that the software system accurately reflects the business domain.

Although these disadvantages exist, using DDD for microservice architectures can still be an excellent way of improving software development outcomes in many scenarios.

**Summary**

In this chapter, you explored microservice architecture and learned about popular patterns such as layered architecture, EDA, and DDD. You learned about the advantages and disadvantages of EDA, when to use it, and how to troubleshoot if things go wrong.

Next, you went through the recommended best practices in the development of microservices. You can leverage the list of tried-and-true best practices in your next project and benefit from them.

Not using architectures such as EDA in a modern-day enterprise is no longer an option. If you continue to use legacy patterns, it is a surefire way for your project and your company to stay stuck in the past and lag behind your competition. A microservice architecture will make your application more scalable, more maintainable, and relevant.

Finally, you further explored how DDD has evolved and can address complex industry use cases to solve business problems. You learned about the principles and components of DDD and used them to design a domain-heavy e-commerce app using AWS services.

Once you build your business app, getting data insight for continuous growth is essential. In the next chapter, you will learn about data lake patterns to get insight from your enterprise data.