**Chapter 8. AWS Orchestration Services**

**A NOTE FOR EARLY RELEASE READERS**

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 12th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

For running systems at large scale, multiple components interact and coordinate with each other to perform any task referred to as orchestration. This communication and coordination between application components can be managed by different AWS services and we can pick and choose our services based on our use-case. In this chapter, we’ll deep dive into core orchestration services such as [Simple Notification Service](https://docs.aws.amazon.com/sns/latest/dg/welcome.html) (SNS) which can be used to broadcast a message to multiple subscribers and [AWS Step Functions](https://aws.amazon.com/step-functions/getting-started/) which can be used for coordination and execution of different AWS or custom services in specific order.

We’ll start off with discussion around AWS services which are helpful in achieving publisher-subscriber design pattern in the AWS Cloud environment such as [Simple Queue Service](https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html) (SQS), SNS & [Amazon Managed Streaming for Apache Kafka](https://docs.aws.amazon.com/msk/latest/developerguide/what-is-msk.html) (MSK) and then later on move on to monitoring, authorization and authentication services such as [CloudWatch](https://aws.amazon.com/cloudwatch/" \t "_blank), [Amazon Cognito](https://aws.amazon.com/cognito/) and [AWS IAM](https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html).

**Amazon Managed Streaming for Apache Kafka**

We discussed message brokers and their architecture in Chapter 8. Apache Kafka is an open-source message broker and event streaming platform used extensively for designing event driven architectures. AWS offers managed service for Apache Kafka to reduce operational overhead on the customer’s end, allowing them to focus more on developing software rather than infrastructure deployment and maintenance of Apache Kafka clusters. Amazon MSK exposes control-plane operations such as create, update & deletion of clusters and data-plane operations such as production and consumption of events to customers.

When [setting up](https://docs.aws.amazon.com/msk/latest/developerguide/getting-started.html) the Amazon MSK cluster,we specify the number and type of broker nodes along with storage capacity in each AZ. A minimum of two AZ must be selected to ensure high cluster availability. Amazon MSK automatically detects broker failures and replaces the node with a healthy node with the same IP address. Similar to multiple other services in the AWS environment, Amazon MSK is also offered in two capacity modes, Serverless and Provisioned. We can choose the Serverless option to let AWS worry about infrastructure configuration while Provisioned allows us to select nodes and storage configurations. Additionally, the coordination and synchronization between the broker nodes is managed via [Zookeeper](https://zookeeper.apache.org/) (discussed in Chapter 5) nodes and Amazon MSK creates these nodes for us.

The next set of configurations in cluster setup are related to networking, security and monitoring. You should select AZs and subnets in which brokers should be launched and then define security measures. First let’s discuss on how clients can access Amazon MSK cluster:

*Unauthenticated access*

As the name suggests, we can allow clients to directly access our cluster without any authentication, though this is not a recommended way to configure access control.

*IAM role based access*

The access to Amazon MSK cluster is managed by IAM roles and associated IAM policies.

*SASL/SCRAM authentication*

[SASL/SCRAM](https://docs.aws.amazon.com/msk/latest/developerguide/msk-password.html) means Simple Authentication and Security Layer/Salted Challenge Response Mechanism. We essentially use sign-in credentials (username and password) for clients and store them to [Amazon Secrets Manager](https://docs.aws.amazon.com/secretsmanager/latest/userguide/intro.html) associated with a secret resource. This provides a secure method of cluster access while Amazon Secrets Manager takes full responsibility for audit, update and rotation of credentials. Additionally, we can share the same credentials across the clusters as necessary.

*TLS Authentication*

We can use TLS authentication for client access and store these certificates in Amazon Certificate Manager ([ACM](https://docs.aws.amazon.com/acm/latest/userguide/acm-overview.html)).

For data security, we can encrypt our data both at rest and in-transit. Encryption at rest can be enabled by AWS managed or customer managed key. For in-transit encryption, we can use TLS encryption for communications within the cluster and use PlainText or TLS Encryption or both for communication between broker and clients.

[Amazon Kinesis](https://aws.amazon.com/kinesis/) is another service which provides data streaming capabilities similar to Amazon MSK and we’ll deep dive into it in the next chapter. AWS also offers SNS and SQS (which we’ll discuss shortly) which can be used for creation of topics for publishing events and the same can be consumed by multiple consumers such as SQS. In Chapter 11, we discussed why customers might prefer EKS instead of ECS for launching their applications. The similar analogy applies here, Kafka is an open source message broker service which can be used in place of SNS-SQS. The reasons you might choose it are similar as well—you could be migrating your workloads to AWS Cloud or running existing workloads on AWS Cloud, or you just love open-source software with full visibility on development features. Let’s move our attention to SQS and SNS now which can be used in place of Amazon MSK for management of topics and queues.

**Amazon Simple Queue Service**

Chapter 1 introduced you to the concept of asynchronous communication and Chapter 7 to Message Queues. SQS is a fully managed AWS queue service which automatically scales as per customer traffic requirements. It is a highly available service which doesn’t require any maintenance and deployment work from the customer’s end. Some key use-cases where SQS stands out are:

*Application decoupling*

Two microservices don’t interact with each other directly and communicate with SQS inbetween. This avoids any service dependency such that one service being down doesn’t impact the other.

*Back pressure control*

The client consuming messages from SQS can consume messages at its own rate. The messages in SQS are kept until a maximum of 14 days or as per customer configurations.

Let’s understand the important concepts and key considerations related to SQS:

*Visibility Timeout*

The message from the queue is not deleted after it is received by one consumer, instead it is kept in SQS owned temporary storage until a timeout setting, referred to as visibility timeout. Once the consumer processes the message, it should inform SQS to delete the message within the visibility timeout. In scenarios set up for no communication till visibility timeout, the message is made available again for other consumers to consume. It can have values from 0 seconds to 12 hours.

*Retention Period*

This is the amount of time the message is kept in the queue if not received by any consumers. It can have values from one minute to 14 days.

*Dead Letter Queue*

There may be scenarios where a consumer failed to process a message but we might need the message for use-cases such as retrying after sometime or for debugging purposes. This is where Dead Letter Queue(DLQ) can help—the consumer can push failed messages to DLQ for any further action at a later point of time.

*Maximum Message Size*

The maximum allowed message size is 256KB. This allowed size should be good enough for all standard use-cases but for scenarios with requirements for more than it, we recommend you consider S3 or DDB. The message can be stored as an S3 or DDB object and the same can be referenced in a SQS message.

*Message Delivery*

Two consumers can’t compete for a single message at the same time, only a single thread can process a message at once. This ensures reliability as multiple producers can send messages and multiple consumers can receive messages.

*SQS Types*

SQS offers two types of queues–Standard and FIFO. Standard queues don’t guarantee message ordering while FIFO guarantees message ordering, refer to Table 12-1 for a detailed comparison.

|  |  |  |
| --- | --- | --- |
| Comparison Parameter | Standard Queue | FIFO Queue |
| Message ordering | Message ordering works on the best effort basis. It is not guaranteed that messages are delivered in the same order they are received by SQS. | Message ordering is guaranteed. |
| Throughput | Unlimited throughput, scales as per customer needs. | Maximum allowed is 300 TPS. Additionally, requests can be batched in a batch of 10 so via this, essentially we can achieve a maximum of 3000 TPS. |
| Message Delivery | Messages might be delivered more than once to clients, standard queue ensures at least-once delivery. The application logic should be idempotent to avoid unexpected behavior. | Messages are delivered exactly once. |
| Cost | Relatively cheaper than FIFO. | [~25% costlier](https://aws.amazon.com/sqs/pricing/) than standard queues. |
| Table 8-1. Standard vs FIFO queues | | |

SQS is widely used with Simple Notification Service(SNS) to meet message broker requirements without actually maintaining any infrastructure such as support use-case of [messaging fanout](https://aws.amazon.com/getting-started/hands-on/send-fanout-event-notifications/). For a single message to be delivered to multiple recipients, it can be published to an SNS topic to which multiple SQS are subscribed to. The copy of the message is sent to each SQS from which it can be independently processed by clients. Let’s dig more into SNS—we’ll discuss what use-cases it solves and how it differs from SQS.

**Amazon Simple Notification Service**

SNS is an intermediary service which enables communication between producers and consumers. A producer essentially publishes a message to a SNS topic and then SNS holds the responsibility to forward this message to all the subscriptions (aka consumers) via push mechanism. The supported consumers are Amazon Kinesis Data Firehose streams, SQS, Lambda, HTTP(S) endpoints, email, mobile push notifications and mobile text messages. Going back to our Cafe Delhi Heights restaurant example from previous chapters, let’s think a little bit about different offers and coupons we get on our phones. As your phone number is saved to the restaurant system and you agree to receive text messages, the restaurant sends messages if there are any new offers or discounts. Here, the restaurant is a message producer which publishes messages to discount channels and we as subscribers get these messages.

Before moving onto important concepts related to SNS, let’s understand the main differences between SQS and SNS via Table 12-2.

|  |  |  |
| --- | --- | --- |
| Comparison Parameter | SQS | SNS |
| Message Delivery mechanism | Application should poll messages from SQS. | SNS pushes the messages to its subscribers. |
| Parallelism | Only a single consumer can access a message at any time and it’s the consumer’s responsibility to delete messages from the queue once processed. | SNS pushes messages to all its consumers in parallel and each consumer can process the message independently. |
| Message Delivery Latency | As it’s the application’s responsibility to poll messages from SQS, it can introduce some latency. | Messages are delivered to consumers as soon as they are published (near real-time) to SNS topics. |
| Table 8-2. SQS and SNS comparison | | |

From the above differences, it’s clear that both SQS and SNS solve different use-cases and one service should not be seen as a replacement to another service. Now, let’s explore more on SNS by looking into important concepts and key considerations related to SNS:

*Message Attributes*

Message attributes are optional metadata items that can be sent along with the message body of messages, represented by key, value and type with supported types being String, String.Array, Binary and Number. These attributes can be used to take appropriate action on a message without processing the message body. Message attributes can optionally be used to filter messages at subscriber level. For example, a SNS topic publishes weather condition messages for the entire Asia continent and it has the country code included in the message attributes. So if the application only wants to process India-specific messages, it can directly filter out other messages without even reading the message.

*Message Filtering*

You saw one example above of how message attributes can be helpful in message filtering. This is achieved via assigning a filter policy to topic subscription, this way only the intended messages are delivered to subscribers. Filter policy can also be applied on the message body, though the message body should be a well-formed JSON object.

*Message Durability*

The message received from the publisher is stored across multiple AZs before acknowledging success. This ensures message availability even during AZ downtimes.

*Message Delivery Failure Handling*

There can be scenarios when a subscriber is not available to receive messages, so SNS has a retry policy based on the [delivery protocol](https://docs.aws.amazon.com/sns/latest/dg/sns-message-delivery-retries.html) to overcome unavailability issues. In case the message is not delivered even after retries, we can configure a dead-letter queue so messages are pushed there and can be polled when our system comes back online.

*Message Security*

To ensure data security or to handle sensitive data, we can [enable Server Side Encryption](https://docs.aws.amazon.com/sns/latest/dg/sns-enable-encryption-for-topic.html) (SSE) on SNS topics by specifying an AWS KMS key. Once enabled, SNS encrypts the received message and stores it in encrypted form. The message is decrypted when it is being forwarded to topic subscribers. SSE only encrypts the message body so we recommend avoiding storing any sensitive information in topic and message metadata.

*Message Ordering*

To cater strict message ordering needs and prevent message duplication, SNS supports [FIFO topics](https://docs.aws.amazon.com/sns/latest/dg/sns-fifo-topics.html) similar to what we discussed in SQS. We can use standard topics by default if there are no such requirements.

A combination of SQS, Lambda and SNS is widely used for multiple use-cases to support event-driven asynchronous architecture and orchestrate simple workflows. For use-cases with requirements to orchestrate complex workflows in event-driven steps, let’s explore AWS’s offered workflow orchestration services.

**Workflow Orchestration**

AWS offers two workflow orchestration services, Step Functions and Amazon Managed Workflow for Apache Airflow (MWAA). Let’s take a simple example of online food ordering at our favorite restaurant–Cafe Delhi Heights. The sequence of steps to place an order after food selection are: process payment, send food order requests to restaurants, choose delivery partners, and track delivery, as shown in [Figure 8-1](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch08.html#fig_1_aws_step_function_food_ordering_workflow). Both the offered services are fully managed by AWS so customers can focus on building their applications without any worry of infrastructure maintenance. Increased application reliability is the main benefit of executing application logic in steps as it ensures independent execution(issues in one component don’t impact others) and proper failure handling.

AWS Step Function Food Ordering Workflow

**Figure 8-1. AWS Step Function Food Ordering Workflow**

AWS Step Functions is an Amazon proprietary service, while Amazon MWAA is built on top of open-source [Apache Airflow](https://airflow.apache.org/). Let’s explore more on both of these options.

**AWS Step Functions**

AWS Step Functions (SFn) is an AWS fully managed serverless and visual workflow orchestration service which helps customers to create and run [state machines](https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-state-machine-structure.html) with ease to coordinate among the distributed applications components. Here are some basic terms you need to know in order to fully understand SFn:

*State Machine*

A state machine represents the orchestration workflow as a sequence of steps, relationship among them and their input & output. These steps coordinate among themselves and interact with different components such as an [Activity](https://docs.aws.amazon.com/step-functions/latest/dg/concepts-activities.html), Lambda Function, SQS, DynamoDB or any [other service](https://docs.aws.amazon.com/step-functions/latest/dg/concepts-service-integrations.html) to complete a specific operation.

*Task*

A single state or step in state machine is referred to as a task, which determines action to be executed, such as calling Lambda to fetch an object from a S3 bucket and parse it. In short, a single state’s responsibility is to take input from the previous state, execute specified work and pass on the output to the next state.

*Activity*

A task holds responsibility to perform some work—the work can be executed by workers on an EC2 machine, ECS, Lambda function or any application with the ability to make HTTP connections hosted anywhere. This is enabled by SFn feature called activity. A SFn workflow step awaits for activity workers to poll SFn for work via [GetActivityTask](https://docs.aws.amazon.com/step-functions/latest/apireference/API_GetActivityTask.html" \t "_blank) API, complete the work and send execution callback to SFn via [SendTaskSuccess](https://docs.aws.amazon.com/step-functions/latest/apireference/API_SendTaskSuccess.html" \t "_blank) or [SendTaskFailure](https://docs.aws.amazon.com/step-functions/latest/apireference/API_SendTaskFailure.html" \t "_blank) API. The timeout value to wait for callback can be configured in task definition as per the nature of application and your use-case.

We use [Amazon States Language (ASL)](https://states-language.net/spec.html) or SFn console to create our workflows. ASL is JSON based structured language to define workflow states declaratively as described in the code snippet:

{

    "Comment": "Amazon States language Example",

    "StartAt": "Hello World",

    "States": {

    "Hello World": {

      "Type": "Task",

      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:HelloWorld",

      "End": true

    }

  }

}

SFn always encrypts the data at rest using transparent server-side encryption and all the data being passed from SFn to any integrated services is encrypted using Transport Layer Security (TLS). This ensures protecting our sensitive data without any operational overhead. The action being taken by the state is determined by the type of state. Here are different state types that we can choose from while designing our state machine:

*Pass*

The [Pass state](https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-pass-state.html) doesn’t perform any work and just passes its input as its output. This state is mostly useful in state machine debugging and can also be used in [transforming input](https://docs.aws.amazon.com/step-functions/latest/dg/input-output-inputpath-params.html) so as transformed data is passed to the following state.

*Task*

As we discussed in previous sections, task essentially performs the work in a state machine. The [Task state](https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-task-state.html) represents a single unit of work via a Lambda function, activity or API actions of different AWS services.

*Choice*

The [Choice state](https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-choice-state.html) is similar to the if-else statement of any programming language. It is a conditional statement that evaluates to true or false to determine further actions to be taken in a state machine.

*Wait*

The [Wait state](https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-wait-state.html) helps in taking a pause action during state machine workflow execution. The timeout value can be a relative value in seconds from state start time or an absolute time as exact timestamp.

*Succeed*

The [Succeed state](https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-succeed-state.html) is terminal state to stop workflow execution successfully.

*Fail*

The [Fail state](https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-fail-state.html) is the opposite of Succeed state and stops the workflow execution with failure. We can also explicitly catch the failure to take any specific action or just ignore failure to move onto the following state.

*Parallel*

The [Parallel state](https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-parallel-state.html) is helpful in adding multiple branches in a state machine. This can be helpful in executing independent states in parallel to make workflow processing faster.

*Map*

The [Map state](https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-map-state.html) is helpful to run a set of workflow steps in parallel for each item in the dataset as a JSON array or CSV file. Map state supports [inline](https://docs.aws.amazon.com/step-functions/latest/dg/concepts-asl-use-map-state-inline.html) and distributed mode of operation and we can select one of them as per our requirements.

* Inline mode supports input as a JSON array from the previous step and can run up to 40 concurrent executions and the execution history should not exceed 25,000 entries.
* Distributed mode can support JSON or CSV files stored in Amazon S3 or a JSON array from the previous step. This mode offers high concurrency as compared to inline mode and supports up to 10,000 parallel executions without any hard limit on execution history.

**NOTE**

The difference between parallel and map state is support for dynamic parallelism. We define multiple branches to be executed in parallel in parallel state while map state can create parallel child branches dynamically based on input.

We should always be prepared for service disruption and to overcome failure scenarios in SFn, we recommend adding [error handling, retry mechanisms](https://docs.aws.amazon.com/step-functions/latest/dg/concepts-error-handling.html) and alerting mechanisms as applicable for better success of the state machine.

The business requirements vary from use-case to use-case, such as event processing volume, system durability, workflow execution time, idempotency, etc. To address these different requirements, SFn is offered in two types–Standard and Express workflows. The detailed comparison of these two types is captured in Table 12-3.

|  |  |  |
| --- | --- | --- |
| Comparison Parameter | Standard Workflows | Express Workflows |
| Primary use-case | Used for long running workloads(up to 1 year), durable and auditable workflows such as ETL workloads. | Used for short lived (up to 5 years) high-volume event-processing workloads such as streaming data processing. |
| Execution model | Workflow is executed exactly once unless explicit retry behavior is in place. | Support is available for both synchronous and asynchronous execution models. The execution is at-least once for async and at-most once for sync workflows. Sync workflows can be invoked from API Gateway, Lambda function or by using [StartSyncExecution](https://docs.aws.amazon.com/step-functions/latest/apireference/API_StartSyncExecution.html" \t "_blank) API call. |
| Support for SFn activities | Supported with a maximum of 1000 pollers allowed on the same activity arn. | Not supported. |
| Execution history | Maximum 25,000 entries in execution event history. To avoid this, we can explore using the Distributed mode Map state or starting a new state machine from Task state of running state machine execution. | No hard limit on the number of events. |
| Table 8-3. Standard and Express workflow comparison | | |

**NOTE**

The type selection should be an informed choice as it can’t be changed after workflow creation.

We can use the AWS SFn console to debug our workflows for failures on any other analysis, such as execution time, input, output, failure reason, retry mechanism, etc. which helps in diagnosing production issues with ease.

**Amazon Managed Workflow for Apache Airflow**

[Apache Airflow](https://airflow.apache.org/) is an open-source tool to programmatically create, schedule and monitor workflows. Amazon MWAA uses message queues to orchestrate an arbitrary number of workers to desired scale, but it also comes with setup and operational management cost. AWS takes this workload off the customer’s plate and offers Amazon MWAA as a managed service to scale as necessary. Here are some features which distinguishes it from open-source tools:

*Scalability*

We can define minimum and maximum of workers for the Amazon MWAA environment and AWS handles the capacity management.

*Security*

We can use IAM roles and policies for different AWS services access management and additionally, all the workers and schedulers run in [Amazon MWAA’s VPC](https://docs.aws.amazon.com/mwaa/latest/userguide/vpc-vpe-access.html) ensuring private fleet management. All data is encrypted by default using AWS KMS as added security. We can enable private or public access modes as necessary by using [AWS IAM](https://docs.aws.amazon.com/mwaa/latest/userguide/access-policies.html) policies and [AWS IAM Identity Center](https://aws.amazon.com/iam/identity-center/). Public access mode ensures accessibility of Apache Airflow server over the internet via VPC endpoint while private access mode is only accessible in your VPC.

*Operations Management*

The setup is very easy with a few clicks in the AWS console and we don’t have to worry about version upgrades or applying any security patches.

*Monitoring*

To monitor and analyze Amazon MWAA tasks and workflows, we can use Apache Airflow logs and Apache Airflow metrics in CloudWatch.

We require logs and metrics capabilities to monitor our own application health as well as the health of AWS resources we’re using to create our infrastructure. Let’s turn our attention to CloudWatch service which offers these features.

**Amazon CloudWatch**

Launching an application on AWS Cloud infrastructure is the first step, but now we have to ensure this application keeps on running, serving customer traffic without any issues. To monitor application health and debug any issues that occur, AWS offers CloudWatch. CloudWatch helps with storage & search capability on application logs, visualizing application metrics & dashboards and setting up alerts on any unexpected application behavior.

**Application Logs**

As applications run to serve customer requests, there are logs generated—these can be custom logs or language runtime specific logs. These logs are helpful in debugging any issue that has occurred in the system. We can publish logs to CloudWatch in real time for any compute platform we’re using and look for specific error logs via CloudWatch search functionality or CloudWatch Insights feature. Here are some key features and terminology associated to CloudWatch logs:

*Logs Storage*

CloudWatch can act as a central repository of all of our application or other AWS services logs. The logs are part of log streams, which are a sequence of log events that share the same source. Further all the log streams with similar properties such as source, retention, monitoring and access control settings are part of a log group. For example, software applications have been set up to create hourly log streams which means every hour a log stream can be created similar to log\_stream\_2023\_05\_15\_07 (log\_stream\_name\_year\_month\_day\_hour).

*Personal Information Identification Data*

We should avoid logging any personal identification information such as social security numbers, medical history, credit card details, etc. This might be pushed to production by mistake and to detect such logs, CloudWatch provides a capability to [identify and mask](https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/mask-sensitive-log-data.html) the log statement by leveraging pattern matching and machine learning.

*Logs Search*

We definitely need a search capability on the top of logs to help with debugging. CloudWatch supports multiple filters by following pattern matching syntax. We can additionally use these filters to identify specific patterns in logs and publish a metric for it such as counting ERROR 400 for a particular API.

CloudWatch offers Log Insights as an additional feature for interactive search, analysis and visualization of logs data. We can search in up to 50 log groups in a single request via supported [query language](https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_QuerySyntax-examples.html). For example, total number of InternalServerException across five microservices log groups.

For general visualization such as CPU utilization or total invocations of an API where we don’t want to run queries to plot graphs on top of data, the effective way is to utilize CloudWatch metrics and CloudWatch alarms can be used for raising alerts for any unexpected behavior.

**Metrics and Alarms**

CloudWatch metrics helps in application monitoring and enables users to search & graph metrics and create alarms on top of them. For example, it can plot service custom metrics such as number of 500 HTTP error codes and raise an alarm if, for a continuous 15 minutes, the total count of 500 HTTP errors per minute is greater than 10. Let’s dig into metrics and alarms in a little more detail.

**Metrics**

A metric is essentially a time-ordered set of data points being sent to CloudWatch for visualization. As mentioned in the previous section, the metrics can be AWS resource metrics or custom application metrics. Most of the AWS services publish some default metrics without charging customers, and we can definitely further tune them as necessary—for example, with minute level granularity. Similar to Log Insights, [CloudWatch Metrics Insights](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/query_with_cloudwatch-metrics-insights.html" \t "_blank) feature can be used to query and find patterns on different published metrics and further create alarms on them. Let’s understand some terminology associated with metrics:

*Namespace*

Namespace is a root level identifier to distinguish metrics from one another. For example, EC2 service metrics are captured in the EC2 namespace. Similarly we can create our custom namespaces, for example, multiple microservices in a single AWS account can be separated by namespaces.

*Dimensions*

Dimension is a metric identification property represented by name-value pairs. A single metric can have up to 30 dimensions associated with it. For example, for representing API latency of specific operation of an application, the namespace is application name with two dimensions as API name and metric type.

*Metric Resolution*

You can choose between standard resolution and high resolution, depending on the granularity required for your use-case. Metrics are available at a minimum of 1-minute granularity in standard resolution or 1-second in the case of high resolution.

*Statistics*

Statistics can be used for data aggregation over a period of time—a couple of examples are maximum latency or sum of API invocations in the last hour. We can choose from a [list of supported statistics](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/Statistics-definitions.html) according to our business needs, such as Maximum, Minimum, SampleCount, Sum, Average, Percentile, etc. Further, each statistic has a [unit associated](https://docs.aws.amazon.com/AmazonCloudWatch/latest/APIReference/API_MetricDatum.html) with it, such as count, percent, seconds, etc.. For example, the average CPU Utilization in the last hour would be shown as 75 percent.

The CloudWatch metrics can also be streamed for further analysis to the destination of our choice in near real-time in JSON or [OpenTelemetry 0.7.0](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch-metric-streams-formats-opentelemetry.html" \t "_blank) format. A single metric stream can include up to 1000 filters which are helpful in streaming selected metrics, giving us control on the metrics we want to stream.

Some use-cases where this could be helpful are:

* Deliver metrics to S3 via Amazon Kinesis Data Firehose Delivery Stream. This can be useful (along with billing data) to look for cost optimization and resource performance.
* Third-party service providers can be chosen as a destination for monitoring and troubleshooting applications.

Metrics are good for visualization and debugging purposes but for raising an alarm or sending a notification for unexpected behavior, we should set up alarms.

**Alarms**

Application monitoring is an important aspect for maintaining application health and alarms help to monitor application metrics and notify users or other applications to take appropriate action when the configured thresholds are breached. For example, you can take auto scaling action by adding one instance when an application’s EC2 CPU utilization is greater than 75 percent and remove one instance when it is less than 30 percent. Let’s discuss the important things to consider when setting up alarms:

*Metric*

An alarm is set up on top of a single metric or a math expression on a combination of metrics. To set up an alarm, it should include all metric information such as metric namespace, metric name and all dimensions.

*Alarm Conditions*

An alarm is invoked when specific conditions are met and these conditions are defined by  parameters such as threshold type, alarm condition, data points to alarm and missing data treatment.

* There are two supported types of threshold configurations: static and anomaly detection.
* Static configuration can be used to configure a specific threshold value on greater than, greater/equal to, lower than and lower/equal to alarm conditions.
* Anomaly detection allows the use of a band of values as a threshold with supported conditions as outside of the band(greater than or less than), lower than and greater than the band.
* We might not always want to raise an alarm with just one metric data point. This is a configurable property for an alarm and can be set up as per alarm requirement.
* There can be scenarios when metric is not pushed to CloudWatch—this can be due to application issues or if there was no actual metric to be published for a specific time period. In both of these cases, we can define how to treat the missing data. The supported behaviors are missing data being treated as good (within the threshold), bad (breaching the threshold), ignored (maintaining current alarm state) and missing (transition alarm to insufficient data for alarm condition evaluation).

*Configurable Actions*

The alarm can be in three states: ok state, in-alarm state or insufficient data to evaluate alarm. The alarm actions can be configured on state transition from one to another. The supported alarm actions are notification, auto scaling action, EC2 action, and systems manager action.

* Notification action allows us to notify people via sending a message to the Simple Notification Service (SNS) topic.
* Auto Scaling action allows us to add or remove instances from an auto scaling group as per alarm threshold.
* EC2 action allows us actions such as rebooting, stopping, terminating or recovering of an EC2 instance.
* We can also set up [Systems Manager](https://docs.aws.amazon.com/systems-manager/latest/userguide/what-is-systems-manager.html) action by creating an [OpsItem](https://docs.aws.amazon.com/systems-manager/latest/APIReference/API_OpsItem.html" \t "_blank) which the operations team can look into and take appropriate action on an alarm.

We can also create an alarm on top of multiple alarms, referred to as *composite alarm*. This can be helpful to reduce operation noise by configuring actions on composite alarms and skipping the same on child alarms. The composite alarm is invoked based on the configured conditions or [rule expressions](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/Create_Composite_Alarm.html) such as reach to alarm state when all child alarms are in alarm state or any one of them in alarm state. For example, you could set up a composite alarm on application resource utilization such as CPU utilization, memory usage and storage space.

The alarms are activated on state change in metric based on a certain threshold. Similar to this, any state in AWS resources is captured via CloudWatch Events.

**CloudWatch Events**

AWS services or resources transition from one state to another state. For example, as we launch an EC2 machine, it transitions from PENDING to RUNNING state after its entire setup and configurations. These state changes can be streamed in near real-time via CloudWatch Events to a desired destination to take any appropriate action. CloudWatch Events can additionally be used to schedule automated actions using cron or rate expressions. For example, we could run a Lambda function everyday at 7 AM to aggregate the previous day’s earnings for our online food delivery restaurant. Some other [target destinations](https://docs.aws.amazon.com/AmazonCloudWatch/latest/events/WhatIsCloudWatchEvents.html) include EC2 instances, Amazon Kinesis Data Streams, ECS tasks, [System Manager Run Command](https://docs.aws.amazon.com/systems-manager/latest/userguide/run-command.html) and [Automation](https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-automation.html), SQS, SNS topics, etc.

AWS offers another service similar to CloudWatch Events called Amazon EventBridge. Both of these services have the same underlying architecture but Eventbridge provides more features to customers. Amazon EventBridge provides backward compatibility to CloudWatch Events APIs and works on a similar concept to CloudWatch Events—it receives an [event](https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-events.html) for state change and applies an associated [rule](https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-rules.html) to route this event to up to five [targets](https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-targets.html).

AWS console automatically redirects to EventBridge page on selection of rules from CloudWatch console. We can also use the CloudWatch Events page, we recommend using EventBridge for more features, such as event ingestion from Software as a Service(SaaS) applications. Let’s discuss on few of these added benefits in more detail:

*Integration with Saas Providers*

Looking at an example of a [PagerDuty](https://www.pagerduty.com/" \t "_blank) application, it can be used to collect data from multiple platforms to provide an unified view to inform team members about any issue occurrence. Let’s say, based on a PagerDuty alert, there is a requirement to restart an EC2 machine. This action can be automated by sending an event to EventBridge and it can deliver that event action to a specific target. If we remove EventBridge from the picture here, polling and custom web hooks are mostly used to fetch data from third party providers to our software application. Polling can be compute-intensive—you might receive empty responses multiple times, whereas web-hooks require communication over the public internet. To address this, Amazon Eventbridge allows integration with [SaaS providers](https://aws.amazon.com/eventbridge/integrations/) over private AWS networks without exposing it to the public internet.

*Custom Event Buses*

Event bus provides a way to communicate from event senders to receivers. All events in CloudWatch Events are routed via the default event bus present in the AWS account, whereas Amazon EventBridge allows creation of [custom event buses](https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-create-event-bus.html) specific to customer workloads and controlled access.

*Enhanced Rules*

Amazon EventBridge allows [content based filtering](https://aws.amazon.com/blogs/compute/reducing-custom-code-by-using-advanced-rules-in-amazon-eventbridge/) to filter values in events. This allows us to apply filtering at the EventBridge level, which helps our application to consume only required event traffic. Additionally we don’t need this filtering logic in our application.

*Schema Registry*

Amazon EventBridge stores schemas for AWS services, integrated SaaS provides and our custom events, helping with inference during our development process.

CloudWatch is an extremely important AWS service ensuring health monitoring of other AWS services and our own custom applications deployed in the AWS Cloud environment. Security of AWS services and resources is another important aspect to consider to avoid any misuse. The next section of this chapter explores AWS IAM service which we can use for access control mechanisms in our AWS account.

**AWS Identity and Access Management**

We touched upon using IAM at multiple places in this book to enable access control and enforce security measures in terms of authentication and authorization for different AWS resources or our overall AWS account. As you create your AWS account for the first time, you log in via root user credentials (with full administrative access) but when accessing different resources or allowing multiple people to login and access an AWS account, we recommend you provide only required access to avoid any mis-use. This permissions granularity is defined via IAM users, groups, roles and policies. For example, a support engineer might only have read-only access for CloudWatch logs but a DevOps engineer might have access to create new log groups or delete existing log groups.

Before we dive deep into more features and benefits of IAM, let’s discuss the basic terminology:

*Root User*

Root user is the user generated as you create your AWS account with email and password credentials. The root user has essentially all the permissions to perform any action in an AWS account and it can’t be denied access by any explicit IAM policies. AWS Organizations [Service Control Policy](https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_policies_scps.html) (SCP) can be used to limit permissions of root user of a member account.

We recommend creating a separate administrative user to perform day-to-day tasks and only use the root credentials for [tasks](https://docs.aws.amazon.com/accounts/latest/reference/root-user-tasks.html) which can only be performed by root user, such as account deletion, sign up for AWS GovCloud(US), register as seller, etc.

*IAM User*

IAM users can represent a person or service with a specific set of permissions to perform a task. IAM users have long term credentials and don’t expire over time. The credentials can be a username & password for logging to AWS console or combination of AWS secret key ID and AWS secret access key for logging programmatically. We recommend avoiding creation of IAM users and use IAM roles with temporary credentials wherever possible. If there is a requirement to create IAM users, we recommend to regularly update passwords and [rotate](https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/automatically-rotate-iam-user-access-keys-at-scale-with-aws-organizations-and-aws-secrets-manager.html) aws secret keys.

*IAM Group*

IAM Groups group users with the same set of permissions. We can’t login to AWS accounts by using group names, but groups make operations easy to manage a similar set of users. This can be helpful in managing users with the same permissions and reduce our burden of assigning the same permissions again and again to users as compared to adding a new user to the group.

*IAM Role*

IAM Role is similar to IAM User but it is not associated with a specific person or service. It is temporarily assumed by a person or service or another AWS account to perform any task. For example, applications running on an EC2 machine will assume a role to make API calls to other resources, such as downloading a file from S3. This ensures temporary access only for the needed time and is automatically revoked after that. IAM Roles can be created with restricted access and this also helps in preventing any unintended changes to the AWS resources environment.

We recommend IAM roles over IAM users with temporary access for a limited period of time. There may be some use-cases where IAM Users will be explicitly required, such as third-party workloads with inability to assume IAM roles, [AWS CodeCommit](https://docs.aws.amazon.com/codecommit/latest/userguide/welcome.html) access, Amazon Keyspaces access or any IAM users created for any emergency issues (we should ensure [Multi-Factor authentication](https://aws.amazon.com/iam/features/mfa/) in these cases).

*IAM Policy*

IAM [Policy](https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html) is a JSON object (apart from [access control lists](https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html#policies_acl) policy type which we discussed in Chapter 10 and is based on [XML structure](https://docs.aws.amazon.com/AmazonS3/latest/userguide/acl-overview.html)) representing a set of permissions which is attached to an IAM identity (user, group or role) or an AWS resource. Consider the example above regarding how EC2 can be provided access to download objects from S3.  Now for that role to have S3 download permissions, we need to attach an IAM policy similar to the code below. The [JSON document of IAM policy](https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html#access_policies-json) essentially contains Effect as Allow or Deny, with Action specifying all the resource operations and Resource specifying determined resources with additional elements such as Condition, Principal and Sid.

{

  "Version": "2012-10-17",

  "Statement": [

{

   "Effect": "Allow",

   "Action": ["s3:ListBucket"],

   "Resource": ["arn:aws:s3:::samplebucket"]

},

{

   "Effect": "Allow",

   "Action": [

     "s3:GetObject"

   ],

   "Resource": ["arn:aws:s3:::samplebucket/\*"]

}

  ]

}

In Chapter 9 we discussed bringing Cafe Delhi Heights online in relation to different AWS storage services. Now you’ve created an AWS account to host and operate the restaurant online and hired many people to help you out. Everyone contributing in development and testing of software applications will require access to the AWS account. There are multiple ways you can provide access:

* Share the root credentials with everyone in the team. We don’t recommend this option, as it provides complete access to everyone. This is not required and might be misused—either knowingly or unknowingly.
* Create IAM users and associate them to groups such as developer-group, devops-group, support-group, etc. This can still be considered a preferred option, but there are some downsides as we discussed in the IAM Role section above.
* You might already be managing the team via some Identity provider (IdP) such as Google, Facebook, Amazon, etc. We can create an IAM Identity Provider entity to establish trust between the IdP and AWS account. This helps all users to assume an IAM role for a limited time and a limited set of permissions without exposing AWS access keys.

As different teams work together, there will always be use cases where one team requires access to an AWS resource in another team’s AWS account, or it could be possible that multiple teams own resources in the same account. For these scenarios, the access can be managed via trust and permissions policies also referred to as *Delegation*. The trust policy specifies trusted account members that are allowed to assume a role, and the permission policy specifies what permissions a trusted user has to perform a task.

AWS IAM helps with securing and managing different AWS resources access. AWS offers another service for user pool management working as a Identity Provider(IdP) and many other features for user authentications. Let’s dig into this service, Amazon Cognito, in the next section.

**Amazon Cognito**

Amazon Cognito is a fully managed highly scalable Customer Identity and Access Management (CIAM) service which helps customers set up and manage their identity pools for authentication (AuthN) and authorization (AuthZ). Amazon Cognito takes the complete ownership of managing compute and storage for supporting this. We can use Amazon Cognito for new user registrations, existing user logins, maintaining guest users and defining access controls. Let’s take a look at a simple example: we want customers of our online restaurant platform to be able to upload images along with food reviews. To allow customers to upload an image directly to our AWS account’s S3 bucket, we can provide temporary write access via Amazon Cognito.

**NOTE**

AuthN establishes the identity of an entity and AuthZ determines what this entity has access to.

In AWS account access terms, AuthN establishes if you can login to an AWS account and AuthZ establishes if you can access a specific AWS resource.

There are certain concepts we should dive deep into for understanding working of Amazon Cognito:

*Identity Providers*

An Identity Provider (IdP) holds responsibility for storage and management of user’s digital identities. Amazon Cognito can itself act as an IdP and we can also integrate it with third-party IdPs (supported protocols are [OAuth 2.0](https://oauth.net/2/), [SAML](https://en.wikipedia.org/wiki/SAML_2.0) and [OIDC](https://en.wikipedia.org/wiki/OpenID)) such as Google, Amazon, Facebook, etc. It is also referred to as Federated Identity since it can be used across the platforms for user authentication. The creation of a trust relationship between AWS and an external identity provider is referred to as Federation. For example, you likely see Google as a sign-in choice on many applications you use day-to-day.

*User Pools*

A User Pool is essentially a user directory which acts as an IdP. The user records are stored in the user pool directory both for users signed in via Cognito or via third-party integrated IdP. It additionally supports all the features such as sign up, forgot password, login, password lengths and policy, enabling MFA, etc.

*Identity Pools*

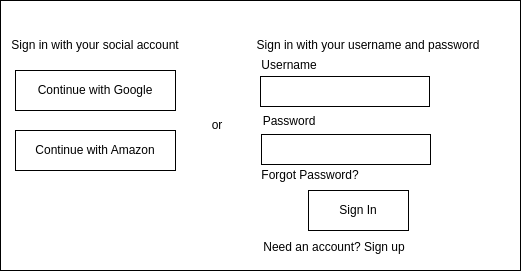
Amazon Cognito Identity Pools help with temporary credentials for application access both for authenticated and non-authenticated users by taking valid tokens such as JWT token or SAML token as input. The temporary role can be assigned to users based on different tags we specify for users. If we look at an example of online food ordering for Cafe Delhi Heights restaurant, the users are allowed to look for food and select food items without logging into the system, but to finally place an order, users must be logged into the system. This can be managed by redirecting to Amazon Cognito Hosted UI once the user clicks on “place order” and then again redirects to payments to confirm order.

*User Pool Hosted UI*

We need some kind of user interface for users to sign up or login to our application. Amazon Cognito offers a URL hosted by AWS for interaction between users and our application. This interface can include email, password for login, or any integrated third-party IdPs as shown in [Figure 8-2](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch08.html#fig_2_amazon_cognito_hosted_ui). Once users are authenticated, they are directed to a specific application page specified as a redirect URL. We can also [customize](https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-pools-app-ui-customization.html) the UI page as necessary, for example with a custom logo or CSS customizations.

*Triggers*

We can consume [Cognito Events](https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-identity-pools-working-with-aws-lambda-triggers.html" \t "_blank) such as user pre-authentication or post-authentication   and run Lambda functions based on them via Amazon Cognito Triggers. The Lambda triggers can help with running extra logic for AuthN and AuthZ of users for underlying applications. For example, we can run a custom set of rules of validation, such as noting that the user location for accessing a particular web page should be India before a user can sign up to the application.



**Figure 8-2. Amazon Cognito Hosted UI**

There are different ways software applications can interact with Amazon Cognito for AuthN and AuthZ:

* Users interact with Amazon Cognito via the frontend application to get a token and use the same token in subsequent API calls. The backend application can interact with Amazon Cognito via AWS SDK Amazon Cognito APIs to validate those tokens against the users.
* Amazon Cognito provides native integration with API gateway without any customer efforts. In case our application APIs are offered via API gateway, we don’t require manual token verification and it can be handled automatically as per the configurations.
* We can also use AWS Amplify to reduce the manual setup at our end and it automatically creates required infrastructure. We can use AWS SDK to interact with Amazon Cognito.

Let’s get back to our example of using Amazon Cognito to assign temporary credentials for uploading an object to Amazon S3. There are essentially two ways of doing this: with Amazon Cognito or by using [S3 pre-signed URLs](https://docs.aws.amazon.com/AmazonS3/latest/userguide/PresignedUrlUploadObject.html). Let’s discuss both of these options in more detail:

*S3 pre-signed URL*

A pre-signed URL can be created with specific permissions on an S3 bucket or a specific object—note that the creator of the pre-signed URL should have the permissions to create it. The one downside of this approach could be pre-defining the object key name, so in order to create pre-signed URLs at runtime, we need some compute layer in between. For example, an AWS Lambda captures file name from a client, then creates a S3 pre-signed URL and responds back to clients to further upload an object.

*Amazon Cognito*

We can use Amazon Cognito’s Identity Pools to assign temporary credentials to users and allow them to directly upload files to S3. The use of this definitely depends on the kind of use-case we’re working on, as this option requires Amazon Cognito setup. It could make our work easy if there is already a setup in place for users AuthN and AuthZ.

The above options can also be used in conjunction as Amazon Cognito authenticates users and generation of pre-signed URL is handled by Lambda. We talked about temporary role assignment based on different tags in Identity Pools—the role can also be assigned based on token and IAM policy, such as you get the role related information as part of the token and you can use the same to assign the role. For requirements with dynamic role creation, [attributes for access control](https://docs.aws.amazon.com/cognito/latest/developerguide/attributes-for-access-control.html) can be helpful in IAM policy.

In the fast pace of development, AWS really helps by providing services such as IAM or Cognito to help with AuthN and AuthZ and we don’t need to build solutions from ground up. Yet another service is AWS AppSync to help frontend and backend teams scale independently in development of APIs at large scale. Let’s look into what AWS AppSync has to offer.

**AWS AppSync**

To fully understand how AWS AppSync works, let’s look at how food items are being displayed to customers on an online food ordering platform. The frontend application essentially displays food items along with their prices, restaurant coupons, food reviews, etc. and this data is maintained by different applications in backend microservices architecture. Here, the frontend application has two options, the first being to invoke APIs of each service and gather data, and the second being to have a single API which collects data from all these different backend services and returns to the frontend.

Recall that we discussed [GraphQL](https://graphql.org/" \t "_blank) in Chapter 6–Communications Networks and Protocols. AWS AppSync is essentially a single GraphQL endpoint that can be used to query multiple databases, microservices and APIs in a single network call. In the above example as well, we can use AWS AppSync to expose a single endpoint to the frontend which can gather data from different microservices. AWS AppSync can also be used for creation of Pub/Sub APIs to publish notifications to subscribed clients such as real-time sports updates via fully managed serverless websocket connections.

Let’s move forward to understand the base components you should be aware of to start with AWS AppSync:

*GraphQL Schema*

GraphQL Schema is essentially a data model specifying the query pattern for data retrieval.

*Resolvers*

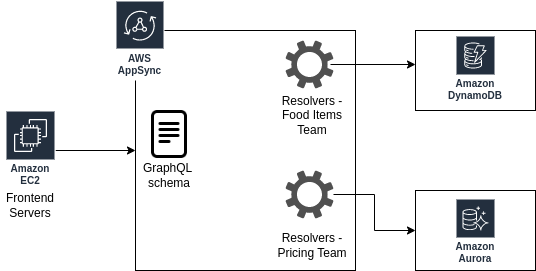
Resolvers are the components that provide a link between schema and different data sources. They are responsible for the conversion of GraphQL payload to underlying system protocol and execute if there are associated IAM permissions.

*Data Sources*

Data Source is the data destination from which Resolvers can fetch the data using the access credentials. For example, DynamoDB, RDS, lambda, HTTP endpoints. The data sources shown in [Figure 8-3](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch08.html#fig_3_appsync_integration_with_multiple_data_sources) can be present in the same AWS account or different AWS accounts.

*Merged API*

A single AppSync API constructed from merge operation of multiple source APIs GraphQL schemas, resolvers and data sources. The source APIs can be managed by independent teams and they can collaborate via Merged API.



**Figure 8-3. AppSync integration with multiple data sources**

Now you might argue, why use AWS AppSync instead of deploying our own GraphQL infrastructure? The reason is fairly simple—using AWS AppSync removes all the operational overhead from the customer’s plate, along with development effort for AuthN & AuthZ, caching, encryption and serving advanced use-cases such as building chat functionality, location aware notifications, live gaming scoreboards, [offline support along with AWS Amplify](https://aws.amazon.com/blogs/mobile/aws-appsync-offline-reference-architecture/), etc. To offer these benefits, AWS AppSync is a fully serverless application which offers [build-time composition of merged APIs](https://d1.awsstatic.com/events/Summits/reinvent2022/FWM316_Accelerate-GraphQL-API-app-development-and-collaboration-with-AWS-AppSync.pdf) and easy integration with AWS services such as AWS WAF, Cloudwatch, Cognito and TLS encryption support.

**Conclusion**

As you know by this point, AWS offers a wide variety of services and it is very important to understand your business requirements to know how to utilize the best services for your goals. For example, a fully managed serverless application such as AWS Step Functions will not give us full visibility around how workflows are orchestrated behind the scenes, but the important question is do we really need to know that? Because AWS SFn is saving us a lot of management and operations costs with additional debugging capabilities via visual workflows.

We’ve also stressed upon application monitoring as an important aspect for ensuring high availability and this can also be achieved by placing metrics, logs and alarms in CloudWatch. We later deep dived into how authentication and authorization services can help in securing our applications and how different components in the AWS infrastructure environment can communicate with each other in a secure and restricted way. We recommend followig the principle of Least Privilege, essentially providing only the minimum set of permissions to application, user or another AWS account that is required to perform an operation.

In the next chapter, we’ll explore how we can launch big data and machine learning workloads on AWS cloud and operate at large scale to serve our customers.