3
The AWS Data Engineer's Toolkit

Back in 2006, Amazon launched **Amazon Web Services** (**AWS**) to offer on-demand delivery of IT resources over the internet, essentially creating the cloud computing industry. Ever since then, AWS has innovated at an incredible pace, continually launching new services and features to offer broad and deep functionality across a wide range of IT services.

Traditionally, organizations built their own big data processing systems in their data centers, implementing commercial or open-source solutions designed to help them make sense of ever-increasing quantities of data. However, these systems were often complex to install, requiring a team of people to maintain, optimize, and update, and scaling these systems was a challenge, requiring significant expenditure on infrastructure and delays while waiting for hardware vendors to install new compute and storage systems.

Cloud computing has enabled the removal of many of these challenges, including the ability to launch fully configured software solutions at the push of a button and have these systems automatically updated and maintained by the cloud vendor. Organizations also benefit from the ability to scale out by adding resources in minutes, all the while only paying for what was used, rather than having to make large upfront capital investments.

Today, AWS offers over 200 different services, including a number of analytics services that can be used by data engineers to build complex data analytic pipelines. There are often multiple AWS services that can be used to achieve a specific outcome, and the challenge for data architects and engineers is balancing the pros and cons of a specific service and evaluating it from multiple perspectives, before determining the best fit for the specific requirements.

In this chapter, we introduce a number of these AWS-managed services commonly used for building big data solutions on AWS, and in later chapters, we will look at how you can architect complex data engineering pipelines using these services. As you go through this chapter, you will learn about the following topics:

- AWS services for ingesting data
- AWS services for transforming data
- AWS services for orchestrating big data pipelines
- AWS services for consuming data
- Hands-on – triggering an AWS Lambda function when a new file arrives in an S3 bucket

Technical requirements

You can find the code files of this chapter in the GitHub repository at the following link: **https://github.com/PacktPublishing/Data-Engineering-with-AWS-2nd-edition/tree/main/Chapter03**.

An overview of AWS services for ingesting data

The first step in building big data analytic solutions is to **ingest data** from a variety of sources into AWS. In this section, we will introduce some of the core AWS services designed to help with this; however, this should not be considered a comprehensive review of every possible way to ingest data into AWS.

Don't feel overwhelmed by the number of services we cover in this section! We will explore approaches to deciding on the right service for your specific use case in later chapters, but it is important to have a good understanding of the available tools upfront.

Amazon Database Migration Service (DMS)

One of the most common ingestion use cases is to sync data from a database system into an analytic pipeline, either landing the data in an Amazon S3-based data lake or in a data warehousing system such as **Amazon Redshift**.

AWS DMS is a versatile tool that can be used to migrate an existing database system into a new database engine, such as migrating an existing **Oracle** database into an **Amazon Aurora PostgreSQL** database. In addition, Amazon DMS can be used to replicate between the same database engine (such as from an on-prem PostgreSQL server to an Amazon Aurora PostgreSQL-compatible server).

When migrating to the same engine, DMS uses the database engine native tools in order to make the migration easy and performant. From an analytics perspective, AWS DMS can also be used to run continuous replication from a number of common database engines into an **Amazon S3 data lake**.

As discussed previously, data lakes are often used as a means of bringing data from multiple different data sources into a centralized location to enable an organization to get the big picture across different business units and functions. As a result, there is often a requirement to perform continuous replication of a number of production databases into Amazon S3.

For our use case, we want to *sync* our production *customer*, *products*, and *order* databases into the data lake. Using DMS, we can do an initial load of data from the databases into S3, specifying the format that we want the file written out in (such as CSV or Parquet), and the specific ingestion location in S3. At the same time, we can also set up a DMS task to do ongoing replication from the source databases into S3 once the full load is completed.

With transactional databases, the rows in a table are regularly updated, such as if a customer changes their address or telephone number. When querying the database using **Structured Query Language** (**SQL**), we can see the updated information, but in most cases, there is no practical method to track changes to the database using only SQL. Because of this, DMS uses the database transaction log files from the database to track updates to rows in the database and writes out the target file in S3 with an extra column added (**Op**) that indicates which operation is reflected in the row – an insert, update, or deletion. The process of tracking and recording these changes is commonly referred to as **Change Data Capture** (**CDC**).

Picture a situation where you have a source table with a schema of **custid**, **lastname**, **firstname**, **address**, and **phone**, and the following sequence of events happens:

- A new customer is added with all fields completed
- The phone number was entered incorrectly, so the record has the phone number updated
- The customer record is then deleted from the database

We would see the following in the CDC file that was written out by DMS:

```
I, 9335, Smith, John, "1 Skyline Drive, NY, NY", 201-555-9012
U, 9335, Smith, John, "1 Skyline Drive, NY, NY", 201-555-9034
D, 9335, Smith, John, "1 Skyline Drive, NY, NY", 201-555-9034
```

The first row in the file shows us that a new record was *inserted* into the table (represented by the `I` in the first column). The second row shows us that a record was *updated* (represented by the `U` in the first column). Finally, the third entry in the file indicates that this record was *deleted* from the table (represented by the `D` in the first column).

We would then have a separate update process that would run to read the updates and apply those updates to the full load, creating a new point-in-time snapshot of our source database. The update process would be scheduled to run regularly, and every time it runs, it would apply the latest updates as recorded by DMS to the previous snapshot, creating a new point-in-time snapshot. We will review this kind of update job and approach in more detail in *Chapter 7, Transforming Data to Optimize It for Analytics*.

Amazon DMS is available either in a provisioned mode (where you select the size of the replication instance used to connect to the source database, do any transformation, and then write to the target) or in a serverless mode (where DMS automatically configures, scales, and manages the resources needed for the migration based on requirements).

When to use: Amazon DMS simplifies migrating from one database engine to a different database engine or syncing data from an existing database to Amazon S3 on an ongoing basis.

When not to use: Amazon DMS does put some load on the production database during migrations, so you need to take this into account.

AWS web page: https://aws.amazon.com/dms/

Amazon Kinesis for streaming data ingestion

Amazon Kinesis is a managed service that simplifies the process of ingesting and processing streaming data in real time, or near real time. There are a number of different use cases that Kinesis can be used for, including ingestion of streaming data (such as log files, website clickstreams, or IoT data), as well as video and audio streams.

Depending on the specific use case, there are a number of different services that you can select from that form part of the overall Kinesis service. Before we go into more detail about these services, here is a high-level overview of the various Amazon Kinesis services:

1. **Kinesis Data Firehose**: Ingests streaming data, buffers for a configurable period, then writes out to a limited set of targets (**S3**, **Redshift**, **OpenSearch Service**, **Splunk**, and others)
2. **Kinesis Data Streams**: Ingests real-time data streams, processing the incoming data with a custom application and low latency
3. **Kinesis Data Analytics**: Reads data from a streaming source and uses SQL statements or **Apache Flink** code to perform analytics on the stream
4. **Kinesis Video Streams**: Processes streaming video or audio streams, as well as other time-serialized data such as thermal imagery and RADAR data

In addition to the four core services listed above that make up the Kinesis service, there is also Kinesis Agent, which can be used to package and send data to Kinesis services.

Amazon Kinesis Agent

AWS provides Kinesis Agent to easily consume data from a system and write that data out in a stream to either Kinesis Data Streams or Kinesis Data Firehose.

Amazon Kinesis Agent is available on GitHub as a Java application under the Amazon Software License (`https://github.com/awslabs/amazon-kinesis-agent`), as well as in a version for Windows (`https://github.com/awslabs/kinesis-agent-windows`).

The agent can be configured to monitor a set of files, and as new data is written to the file, the agent buffers the data (configurable for a duration of between 1 second and 15 minutes) and then writes the data to either Kinesis Data Streams or Kinesis Data Firehose. The agent handles retry on failure, as well as file rotation and checkpointing.

An example of a typical use case is a scenario where you want to analyze events happening on your website in near real time. Kinesis Agent can be configured to monitor the Apache web server log files on your web server, convert each record from the Apache access log format into JSON format, and then write records out reflecting all website activity every 30 seconds to Kinesis, where Kinesis Data Analytics can be used to analyze events and generate custom metrics based on a tumbling 5-minute window.

When to use: Amazon Kinesis Agent is ideal for when you want to stream data to Kinesis that is being written to a file in a separate process (such as log files).

When not to use: If you have a custom application where you want to emit streaming events (such as a mobile application or IoT device), you may want to consider using the **Amazon Kinesis Producer Library** (**KPL**), or the **AWS SDK**, to integrate sending streaming data directly from your application.

Amazon Kinesis Firehose

Amazon Kinesis Firehose is designed to enable you to easily ingest data in near real time from streaming sources and write out that data to common targets, including Amazon S3, Amazon Redshift, Amazon OpenSearch Service, as well as many third-party services (such as **Splunk**, **Datadog**, and **New Relic**).

With Kinesis Firehose, you can easily ingest data from streaming sources, process or transform the incoming data, and deliver that data to a target such as Amazon S3 (among others). A common use case for data engineering purposes is to ingest website clickstream data from the Apache web logs on a web server and write that data out to an S3 data lake (or a Redshift data warehouse).

In this example, you could install Kinesis Agent on the web server and configure it to monitor the Apache web server log files. Based on the configuration of the agent, at a regular schedule, the agent will write records from the log files to the Kinesis Firehose endpoint.

The Kinesis Firehose endpoint would buffer the incoming records, and either after a certain time (1-15 minutes) or based on the size of incoming records (1 MB-128 MB), it would write out data to the specified target. Kinesis Firehose requires you to specify both a size and a time limit, and whichever is reached first will trigger the writing out of the file.

When writing files to Amazon S3, you also have the option to transform incoming data into **Parquet** or **ORC** format or perform custom transformations of the incoming data stream using an Amazon Lambda function. Kinesis Data Firehose also supports **dynamic partitioning**, enabling you to specify a custom partitioning configuration. With dynamic partitioning, as your data is written to your S3-based data lake it can be partitioned based on custom partition keys contained in the data payload.

When to use: Amazon Kinesis Firehose is the ideal choice for when you want to receive streaming data, buffer that data for a period, and then write the data to one of the targets supported by Kinesis Firehose (such as Amazon S3, Amazon Redshift, Amazon OpenSearch Service, an HTTP endpoint, or a supported third-party service).

When not to use: If your use case requires very low-latency processing of incoming streaming data (that is, immediate reading of received records) or you want to use a custom application to process your incoming records or deliver records to a service not supported by Amazon Kinesis Firehose, then you should consider using **Amazon Kinesis Data Streams** or Amazon **Managed Streaming for Apache Kafka** (**MSK**) instead.

AWS web page: https://aws.amazon.com/kinesis/data-firehose/

Amazon Kinesis Data Streams

While Kinesis Firehose buffers incoming data before writing it to one of its supported targets, Kinesis Data Streams provides increased flexibility for how data is consumed and makes the incoming data available to your streaming applications with very low latency (AWS indicates data is available to consuming applications within as little as 70 milliseconds of the data being written to Kinesis).

Companies such as Netflix use Kinesis Data Streams to ingest terabytes of log data every day, enriching their networking flow logs by adding in additional metadata, and then writing the data to an open-source application for performing near real-time analytics on the health of their network.

You can write to Kinesis Data Streams using Kinesis Agent, or you can develop your own custom applications using the AWS SDK or the **Amazon KPL**, a library that simplifies writing data records with high throughput to a Kinesis data stream.

Kinesis Agent is the simplest way to send data to Kinesis Data Streams if your data can be supported by the agent (such as when writing out log files), while the AWS SDK provides the lowest latency, and the Amazon KPL provides the best performance and simplifies tasks such as monitoring and integration with the **Kinesis Client Library** (**KCL**).

There are also multiple options available for creating applications to read from your Kinesis data stream, including the following:

- Using other Kinesis services (such as Kinesis Firehose or Kinesis Data Analytics).
- Running custom code using the AWS Lambda service (a serverless environment for running code without provisioning or managing servers).
- Setting up a cluster of Amazon EC2 servers to process your streams. With this approach, you can use the KCL to handle many of the complex tasks associated with using multiple servers to process a stream,

such as load balancing, responding to instance failures, checkpointing records that have been processed, and reacting to resharding (increasing or decreasing the number of shards used to process streaming data).

Kinesis Data Streams supports two capacity modes. With **Provisioned** mode, you need to specify the number of shards for the data stream upfront, and can then resize the number of shards based on changing requirements. With **On-Demand** mode, you do not need to do any capacity planning or sizing of the Kinesis cluster, as the cluster automatically scales to meet throughput requirements.

When to use: Amazon Kinesis Data Streams is ideal for use cases where you want to process incoming data as it is received, or you want to create a high-availability cluster of servers to process incoming data with a custom application.

When not to use: If you have a simple use case that requires you to write data to specific services in near real time, you should consider Kinesis Data Firehose if it supports your target destination. If you are looking to migrate an existing Apache Kafka cluster to AWS, then you may want to consider migrating to Amazon MSK. If Apache Kafka supports third-party integration that would be useful to you, you may want to consider Amazon MSK.

AWS web page: `https://aws.amazon.com/kinesis/data-streams/`

Amazon Kinesis Data Analytics

Amazon Kinesis Data Analytics simplifies the process of processing streaming data using an Apache Flink application.

An example of a use case for Kinesis Data Analytics is to analyze incoming clickstream data from an e-commerce website to get near-real-time insight into the sales of a product. In this use case, an organization may want to know how the promotion of a specific product is impacting sales to see whether the promotion is effective, and Kinesis Data Analytics can enable this using a Flink application to process records being sent from their web server clickstream logs. This enables the business to quickly get

answers to questions such as "*how many sales of product x have there been in each 5-minute period since our promotion went live?*"

When to use: If you want to use Apache Flink to analyze data or extract key metrics over a rolling time period, Kinesis Data Analytics significantly simplifies this task. If you have an existing Apache Flink application that you want to migrate to the cloud, consider running the application using Kinesis Data Analytics.

AWS web page: https://aws.amazon.com/kinesis/data-analytics/

Amazon Kinesis Video Streams

Amazon Kinesis Video Streams can be used to process time-bound streams of unstructured data such as video, audio, and RADAR data.

Kinesis Video Streams takes care of provisioning and scaling the compute infrastructure that is required to ingest streaming video (or other types of media files) from potentially millions of sources. Kinesis Video Streams enables playback of video for live and on-demand viewing and can be integrated with other Amazon API services to enable applications such as computer vision and video analytics.

Appliances such as video doorbell systems, home security cameras, and baby monitors can stream video through Kinesis Video Analytics, simplifying the task of creating full-featured applications to support these appliances.

When to use: When creating applications that use a supported source, Kinesis Video Streams significantly simplifies the process of ingesting streaming media data and enabling live or on-demand playback.

AWS web page: https://aws.amazon.com/kinesis/video-streams

> **A note about AWS service reliability**
>
> AWS services are known to be extremely reliable, and generally significantly exceed the uptime and reliability of what most organizations can achieve in their own data centers.

> However, as Werner Vogels (Amazon's CTO) has been known to say, *"Everything fails all the time."*
>
> In November 2020, the Amazon Kinesis service running out of data centers in the Northern Virginia Region (us-east-1) experienced a period of a number of hours where there were *increased error rates* for users of the service. During this time, many companies reported having their services affected, including Roomba vacuum cleaners, Ring doorbells, The Washington Post newspaper, Roku, and others.
>
> This is a clear reminder that while AWS services generally offer extremely high levels of availability, if you require absolutely minimal downtime, you need to design the ability to fail-over to a different AWS Region in your architecture.

Amazon MSK for streaming data ingestion

Apache Kafka is a popular open-source distributed event streaming platform that enables an organization to create high-performance streaming data pipelines and applications, and **Amazon MSK** is a managed version of Apache Kafka available from AWS.

While Apache Kafka is a popular choice for organizations, it can be a challenge to install, scale, update, and manage in an on-premises environment, often requiring specialized skills. To simplify these tasks, AWS offers Amazon MSK, which enables an organization to deploy an Apache Kafka cluster with a few clicks in the console and reduces the management overhead by automatically monitoring cluster health and replacing failed components, handling OS and application upgrades, deploying in multiple availability zones, and providing integration with other AWS services.

With Amazon MSK, you need to specify compute and storage capacity for the cluster; however, Amazon MSK is also offered as a serverless cluster type. With this option, you do not need to specify and manage cluster capacity, as **Amazon MSK Serverless** automatically deploys and scales the required compute and storage. To decide between provisioned and

serverless, you need to compare the pricing for your use case, but generally, MSK Serverless is best suited for unpredictable workloads where there are spikes and troughs in streaming throughput.

When to use: Amazon MSK is an ideal choice if your use case is a replacement for an existing Apache Kafka cluster, or if you want to take advantage of the many third-party integrations from the open-source Apache Kafka ecosystem. As Amazon MSK is a managed version of the open-source Apache Kafka solution, if having an open-source solution that enables easy migration to non-AWS environments is important to you, then Amazon MSK may be a good choice.

When not to use: Amazon Kinesis may be a preferred streaming solution if you are creating a new solution from scratch and you are looking for the best integration with other AWS services.

AWS web page: `https://aws.amazon.com/msk/`

Amazon AppFlow for ingesting data from SaaS services

Amazon AppFlow can be used to ingest data from popular SaaS services (such as Salesforce, Google Analytics, Microsoft SharePoint Online, and many more), and to transform and write the data out to common analytic targets (such as Amazon S3, Amazon EventBridge, and Amazon Redshift, as well as being able to write to some SaaS services).

For example, AppFlow can be used to ingest lead data from **Marketo**, a developer of marketing automation solutions, where your organization may capture details about a new lead. Using AppFlow, you can create a flow that will automatically create a new **Salesforce** contact record whenever a new Marketo lead is created.

From a data engineering perspective, you can create flows that will automatically write out new opportunity records created in Salesforce into your S3 data lake or Redshift data warehouse, enabling you to join those opportunity records with other datasets to perform advanced analytics.

AppFlow can be configured to run on a schedule or in response to specific events (for certain sources), and can filter, mask, and validate data, and perform calculations from data fields in the source. AWS also regularly adds additional integrations to the AppFlow service, and, in November 2022, it announced 22 new data connectors, bringing the total number of connectors to over 50.

In addition to connectors for a number of AWS services, there are connectors for third-party services such as Datadog, Facebook, Google Analytics, GitHub, LinkedIn, Salesforce, SAP OData, ServiceNow, Slack, Snowflake, Stripe, QuickBooks, Zoom, and many others.

For details on all AppFlow connectors, review the AppFlow documentation at `https://docs.aws.amazon.com/appflow/latest/userguide/app-specific.html`.

When to use: Amazon AppFlow is an ideal choice for ingesting data into AWS if one of your data sources is a supported source.

When not to use: Amazon AppFlow has limited functionality for transforming data that is ingested. If you require more complex transformations as part of ingestion, or if the data sources you require are not supported, consider similar third-party toolsets (such as Hevo Data), or consider building a custom ingest pipeline using AWS Glue.

AWS web page: `https://aws.amazon.com/appflow/`

AWS Transfer Family for ingestion using FTP/SFTP protocols

AWS Transfer Family provides a fully managed service that enables file transfers directly into and out of Amazon S3 using common file transfer protocols, including **FTP**, **SFTP**, FTPS, and AS2.

Many organizations today still make use of these protocols to exchange data with other organizations. For example, a real-estate company may receive the latest **Multi-Listing Service** (**MLS**) files from an MLS provider via SFTP. In this case, the real-estate company will have configured a

server running SFTP and created an SFTP user account that the MLS provider can use to connect to the server and transfer the files.

With AWS Transfer for SFTP, the real-estate company could easily migrate to the managed AWS service, replicating the account setup that exists for their MLS provider on its on-premises server with an account in its AWS Transfer service. With little to no change on the side of the provider, when future transfers are made via the managed AWS service, these would be written directly into Amazon S3, making the data immediately accessible to data transformation pipelines created for the Amazon S3-based data lake.

When to use: If an organization currently receives data via FTP, SFTP, FTPS, or AS2, it should consider migrating to the managed version of this service offered by Amazon Transfer.

When not to use: There are other AWS Marketplace options for running a managed file transfer service, so you should compare options based on both features and pricing in order to determine the best match for your requirements.

Some of these other services, such as SFTP Gateway (see `https://help.thorntech.com/`), offer options across multiple clouds, so if multi-cloud is important to you, you may prefer a third-party service.

AWS web page: `https://aws.amazon.com/aws-transfer-family/`

AWS DataSync for ingesting from on premises and multicloud storage services

There is often a requirement to ingest data from existing on-premises storage systems, and **AWS DataSync** simplifies this process while offering high performance and stability for data transfers.

Network File System (NFS) and **Server Message Block (SMB)** are two common protocols that are used to allow computer systems to access files stored on a different system. With DataSync, you can easily ingest and replicate data from file servers that use either of these protocols. DataSync also supports ingesting data from on-premises object-based

storage systems that are compatible with core **AWS S3 API** calls, as well as from **Hadoop Distributed File System** (**HDFS**), commonly used for on-premises data lakes, and from other cloud object stores (such as Azure Blob Storage and Google Cloud Storage).

DataSync can write to multiple targets within AWS, including Amazon S3 and Amazon EFS, making it an ideal way to sync data from on-premises storage, on-premisis data lakes, as well as other cloud services, into your AWS S3-based data lake. For example, if you have a solution running in your data center that writes out end-of-day transactions to a file share, DataSync can ensure that the data is synced to your S3 data lake. Another common use case is to transfer large amounts of historical data from an on-premises system or HDFS data lake into your S3 data lake.

When to use: AWS DataSync is a good choice when you're looking to ingest current or historical data from compatible on premises storage systems, or other cloud storage services, to AWS over a network connection.

When not to use: For very large historical datasets where sending the data over a network connection is not practical, you should consider using the Amazon Snow family of devices.

AWS web page: `https://aws.amazon.com/datasync/`

The AWS Snow family of devices for large data transfers

For use cases where there are very large datasets that need to be ingested into AWS, and either a good network connection is lacking or just the sheer size of the dataset makes it impractical to transfer them via a network connection, the **AWS Snow family of devices** can be used.

The AWS Snow family of devices are ruggedized devices that can be shipped to a location and attached to a network connection in the local data center. Data can be transferred over the local network and the device is then shipped back to AWS, where the data will be transferred to Amazon S3. All the devices offer encryption of data at rest, as well as high-security features such as tamper-resistant enclosures and **Trusted**

Platform Modules (**TPMs**) that can detect unauthorized modifications to the hardware, software, or firmware of the devices.

The Snow devices also offer compute ability, enabling edge computing use cases (edge computing is where you run applications closer to a user, so outside of the core AWS cloud environment). An example edge computing use case is for a factory in a remote area without good internet connectivity, where they can use a Snow device in the factory to collect and process factory IoT data (such as using machine learning models running on the Snow device to predict maintenance issues based on the IoT data).

There are multiple devices available for different use cases, as summarized here:

1. **AWS Snowcone**: Lightweight (4.5 lb/2.1 kg) device with 8-14 TB of usable storage, 2 vCPUs, and 4 GB of RAM for the compute
2. **AWS Snowball Edge**: Medium-weight (49.7 lb/22.5 kg) device with up to 104 vCPUs, 416 GB of RAM, and 210 TB of usable SSD storage

The Snowball Edge devices are available either as **compute-optimized** or **storage-optimized**. The compute-optimized device can have up to 28 TB of SSD storage and includes the option of having an NVIDIA Tesla V100 GPU for compute-intensive applications (such as machine learning or video analysis). The storage-optimized device can have up to 210 TB SSD of storage but does not provide an option for GPU compute.

When to use: The AWS Snow family of devices is ideal for migrating very large volumes of data from on-premises or remote locations to Amazon S3 and can also be used for edge computing use cases and for data migration from locations that have poor internet connectivity.

When not to use: You need to evaluate the available bandwidth from the source environment where you need to migrate data from in order to determine whether a solution such as DataSync may be able to transfer your data quicker than using a Snow device. See the *Best practices for moving your data to AWS* video from *re:Invent 2021*, available at `https://youtu.be/9r9PmMpJGKg?t=219`, for a comparison of DataSync and Snow devices for large transfers.

AWS web page: https://aws.amazon.com/snow/

AWS Glue for data ingestion

AWS Glue provides a serverless Apache Spark environment where you can create code-based solutions for both data ingestion and transformation.

AWS Glue includes a number of built-in connectors for connecting to certain AWS and third-party services, including Amazon RDS, Amazon Redshift, Amazon DocumentDB, MongoDB, MongoDB Atlas, and various JDBC-accessible sources.

In addition, AWS Glue makes it possible to subscribe to a number of connectors from the AWS Marketplace. These include connectors created by AWS and available at no charge, such as the **Google BigQuery connector** (https://aws.amazon.com/marketplace/pp/prodview-sqnd4gn5fykx6?ref_=beagle&applicationId=GlueStudio). It also includes connectors from third parties that may have associated licensing costs, such as the **CData Connector for Salesforce** (https://aws.amazon.com/marketplace/pp/prodview-sonh33r7xavhw?sr=0-11&ref_=beagle&applicationId=GlueStudio).

You can use these connectors when creating AWS Glue jobs in order to connect to a diverse set of sources and to import data as part of your AWS Glue job.

When to use: AWS Glue enables you to import data into AWS as part of a Glue job, using pre-built connectors. This is useful when there is no AWS-managed service for importing data from the required source, or when you want to import data as part of a Glue-based ETL job. The AWS Marketplace includes a number of these pre-built connectors, some at no charge, and others that have a subscription-based cost.

When not to use: AWS Glue uses a Spark-based ETL job to ingest data from a variety of sources, but if you are looking to just ingest data without needing to do Spark-based transformations, then consider the AWS ingestion-focused managed services for your use case (such as Amazon AppFlow or AWS DMS).

Learn more about Glue connectors:
https://docs.aws.amazon.com/glue/latest/ug/connectors-chapter.html

Having examined a number of options for ingesting data into your AWS environment, we will now move on to an overview of services for transforming your data.

An overview of AWS services for transforming data

Once your data is ingested into an appropriate AWS service, such as Amazon S3, the next stage of the pipeline needs to **transform** the data to optimize it for analytics and to make it available to your data consumers.

Some of the tools we discussed in the previous section for ingesting data into AWS can perform light transformations as part of the ingestion process. For example, Amazon DMS can write out data in Parquet format (a format optimized for analytics), as can Kinesis Firehose. However, heavier transformations are often required to fully optimize your data for a differing set of analytic tasks and diverse data consumers, and in this section, we will examine some of the core AWS services that can be used for this.

AWS Lambda for light transformations

AWS Lambda provides a serverless environment for executing code and is one of AWS's most popular services. You can trigger your Lambda function to execute your code in multiple ways, including through integration with other AWS services, and you only pay for the duration that your code executes, billed in 1-millisecond increments, and based on the amount of memory that you allocate to your function. Lambda can scale from a few executions per day to thousands of executions per second.

In the data engineering world, a common use case for Lambda is for performing validation or light processing and transformation of incoming data. For example, if you have incoming CSV files being sent by one of your partners throughout the day, you can trigger a Lambda function to run every time a new file is received, have your code validate that the file

is a valid CSV file, perform some computation on one of the columns and update a database with the result, and then move the file into a different bucket where a batch process will later process all files received for the day.

Another use case is to process an incoming stream of data. For example, you can use Lambda along with Amazon Kinesis Data Streams or Kinesis Data Firehose to process real-time streaming data from a mobile application.

With the ability to run for up to 15 minutes, and with a maximum memory configuration of 10 GB, it is possible to do more advanced processing as well. For example, you may receive a ZIP file containing hundreds of small XML files, and you can process that with a Lambda function that unzips the file, validates each XML file to ensure that it is valid XML, performs calculations on fields in the file to update various other systems, concatenates the contents of all the files, and then writes that out in Parquet format in a different zone of your data lake.

Lambda is also massively parallelized, meaning that it can easily scale for highly concurrent workloads. In the preceding example of processing ZIP files as they arrive in an S3 bucket, if hundreds of ZIP files were all delivered within a period of just a few seconds, a separate Lambda instance would be spun up for each file, and AWS would automatically handle the scaling of the Lambda functions to enable this. By default, you can have 1,000 concurrent Lambda executions within an AWS Region for your account, but you can work with AWS support to increase this limit to the tens of thousands.

AWS Lambda supports many different languages, including **Python**, which has become one of the most popular languages for data engineering-related tasks. In the hands-on activity part of this chapter, we will create a Lambda function that is automatically triggered when a new file is uploaded to a specific S3 location.

AWS Glue for serverless data processing

AWS Glue has multiple components that could have been split into multiple separate services, but these components can all work together, so

AWS has grouped them together into the AWS Glue family. In this section, we will look at the core Glue components related to data processing.

Serverless ETL processing

There are a number of common code-based engines that are popular for performing data-engineering-related tasks. These include:

1. **Apache Spark** is an open-source engine for the distributed processing of large datasets across a cluster of compute nodes, which makes it ideal for taking a large dataset, splitting the processing work between the nodes in the cluster, and then returning a result. As Spark does all processing in memory, it is highly efficient and performant and has become the tool of choice for many organizations looking for a solution for processing large datasets.
2. **Python**, which traditionally runs on a single node, has become an extremely popular language for performing data-science-/data-engineering-related tasks on small to medium-sized datasets.
3. **Ray.io** is an open-source framework that enables running Python code over multiple compute nodes, enabling Python code to be used to process much larger datasets.

The following diagram depicts two of the Glue engines – a single-node Glue Python shell on the left, and a multi-node Glue Apache Spark cluster on the right:

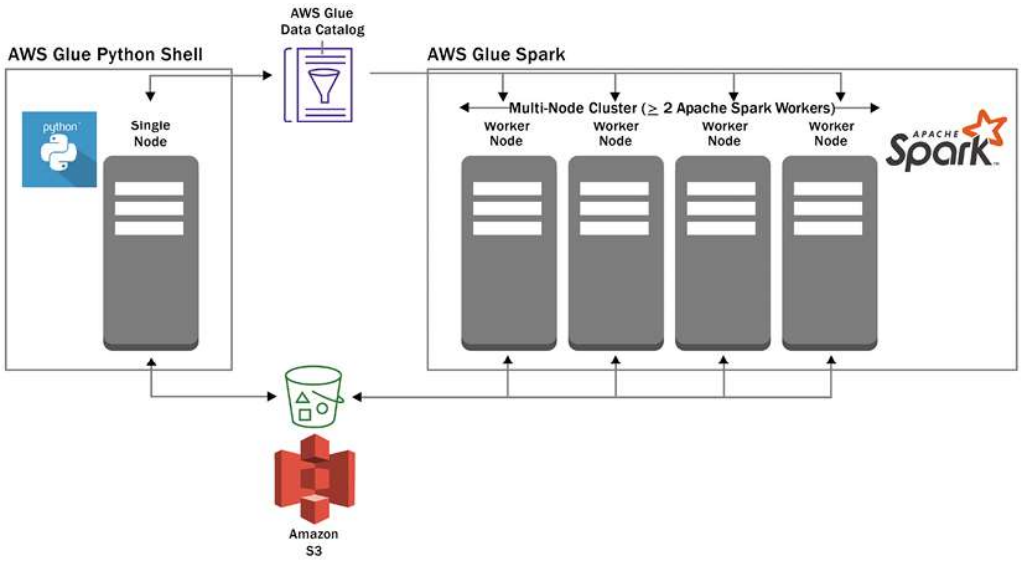

Figure 3.1: Glue Python shell and Glue Spark engines

Both engines can work with data that resides in Amazon S3 and with the **AWS Glue Data Catalog**. Both engines are serverless from the perspective of a user, meaning a user does not need to deploy or manage servers; a user just needs to specify the number of **Data Processing Units (DPUs)** that they want to power their job. Glue ETL jobs are charged based on the number of DPUs configured, as well as the amount of time for which the underlying code executes in the environment.

While AWS Glue does provide additional Spark libraries and functionality to simplify some common ETL tasks, their use is optional, and existing open-source Spark code can be easily run in AWS Glue. AWS Glue also supports **Spark Streaming**, an extension of the core Spark API designed to process live data streams.

You can write new Spark code directly (or use existing Spark code) with the AWS Glue ETL service or you can generate Spark code through a GUI-based tool with the **Glue Studio Visual Editor**. In *Chapter 7*, *Transforming Data to Optimize for Analytics*, we will get hands-on with Glue Studio to join two datasets.

AWS Glue DataBrew

AWS Glue DataBrew is another serverless visual data preparation tool that lets you easily apply transformations to your data, without needing to write or manage any code. DataBrew includes over 250 built-in data transformations, which can be easily assembled via the DataBrew UI to create a *DataBrew recipe*, enabling you to apply multiple transformations to a dataset.

DataBrew includes functionality for both profiling data (gathering statistics on the different columns in the dataset) and for monitoring data quality. It also includes many different types of transformations, such as formatting data, obfuscating PII data, splitting or joining columns, converting timezones, detecting and removing outliers, and many more. For example, you can run a DataBrew profile job that can detect PII data, and then create a DataBrew job that can mask, encrypt, or hash sensitive data.

DataBrew is commonly used by data analysts and data scientists to clean and prepare data for additional processing. In *Chapter 8*, *Identifying and Enabling Data Consumers*, we will do a hands-on exercise on using Glue DataBrew to transform some data.

AWS Glue Data Catalog

To complement the ETL processing functionality described previously, AWS Glue also includes a *data catalog* that can be used to provide a logical view of data stored physically in the storage layer. Objects (such as databases and tables) in the catalog can then be directly referenced from your ETL code.

Glue Data Catalog is a **Hive metastore**-compatible catalog, meaning that it can be used with any system that is able to work with a Hive metastore. As discussed in *Chapter 2*, under the heading *Catalog and search layer*, you get two types of catalogs – business and technical. The Hive metastore, and therefore Glue Data Catalog, is a *technical* catalog.

As an example, if you used **AWS DMS** to replicate your **Human Resources** (**HR**) database in S3, you would end up with a prefix (directory) in S3 for each table from the source database. In this directory, we would generally find multiple files containing the data from the source table – for example, 20 CSV files containing the rows from the source `employee` table.

Glue Data Catalog can provide a logical view of this dataset and capture additional metadata about the dataset. For example, the data catalog consists of a number of databases at the top level (such as the **HR** database), and each database contains one or more tables (such as the `employee` table).

In addition, each table in the catalog contains metadata, such as the column headings and data types for each column (such as `employee_id`, `lastname`, `firstname`, `address`, and `dept`), a reference to the S3 location for the data that makes up that table, and details on the file format (such as `CSV`).

In the following screenshot, we see a bucket that contains objects under the prefix `hr/employees` and a number of CSV files that contain data imported from the employee database:

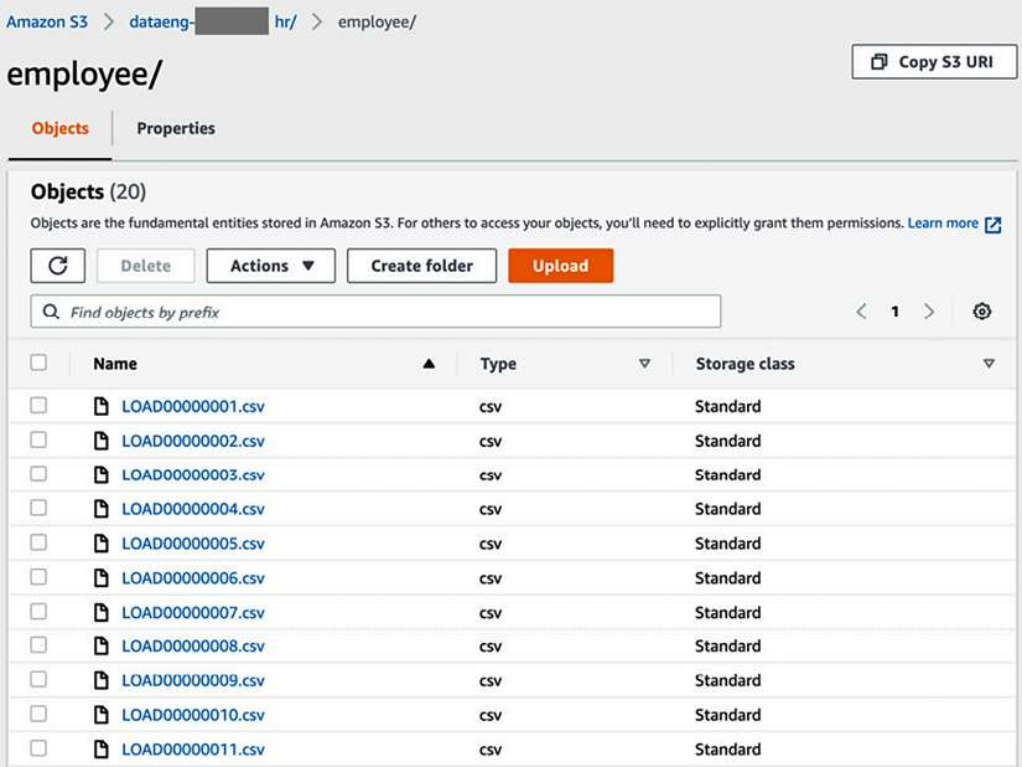

Figure 3.2: Amazon S3 bucket with CSV files making up the employee table

The screenshot of the following AWS Glue Data Catalog shows us the logical view of this data. We can see that this is the `employee` table and it references the S3 location shown in the preceding screenshot. In this logical view, we can see that the `employee` table is in the HR database, and we can see the columns and data types that are contained in the CSV files.

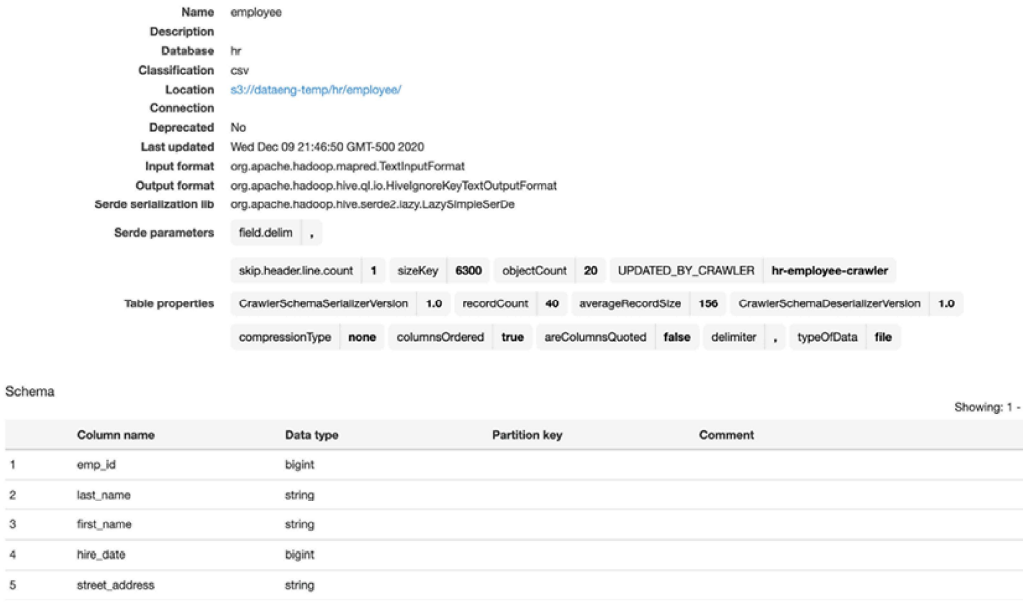

Figure 3.3: AWS Glue Data Catalog showing a logical view of the employee table

Within the AWS ecosystem, a number of services can use AWS Glue Data Catalog. For example, Amazon Athena uses AWS Glue Data Catalog to enable users to run SQL queries directly on data in Amazon S3, and Amazon EMR and the AWS Glue ETL engine use it to enable users to reference catalog objects (such as databases and tables) directly in their ETL code.

AWS Glue crawlers

AWS Glue crawlers are processes that can examine a data source (such as a path in an S3 bucket) and automatically infer the schema and other information about that data source so that AWS Glue Data Catalog can be automatically populated with relevant information.

For example, we could point an AWS Glue Crawler at the S3 location where DMS replicated the *employee* table of our *HR* database. When the Glue Crawler runs, it examines a portion of each of the files in that location, identifies the file type (CSV or Parquet), uses a classifier to infer the schema of the file (column headings and types), and then adds that information into a database in Glue Data Catalog.

Note that you can also add databases and tables to Glue Data Catalog using the Glue API or via SQL statements in Athena, so using Glue crawlers to automatically populate the catalog is optional. In *Chapter 6*, *Ingesting*

Batch and Streaming Data, we will do a hands-on exercise to configure the Glue Crawler to add newly ingested data to Glue Data Catalog.

Amazon EMR for Hadoop ecosystem processing

Amazon EMR provides a managed platform for running popular open-source big data processing tools, such as **Apache Spark**, **Apache Hive**, **Apache Hudi**, **Apache HBase**, **Presto**, **Pig**, and others. **Amazon EMR** takes care of the complexities of deploying these tools and managing the underlying clustered compute resources.

Like many other AWS services, Amazon EMR can run in a provisioned mode (where you specify specific compute resources to use), or it can be run in a serverless mode.

With EMR provisioned mode, you can create an EMR cluster on EC2 nodes, specifying the specific instance types you want to use, and many other detailed configuration parameters (big data applications you want to deploy, automatic scaling options, networking, and more). This option provides you the most control and flexibility for cluster and application configuration, enabling you to fine-tune the performance of your applications. However, this does require you to build up expertise to understand the configuration options and their impact on application performance, such as the optimal EC2 instance type to use for different applications.

Alternatively, you can select to run Spark workloads using **Amazon EMR** on an **Elastic Kubernetes Service** (**EKS**) cluster. If your organization already makes use of EKS to run applications, you can have EMR automatically deploy containers to run Spark workloads using the EKS compute. EMR does not provision the EKS cluster (it only provisions the Spark-based container to run on the cluster), so you are required to have an existing cluster and the skills to maintain and configure EKS.

EMR Serverless provides you with an option to run certain workloads without needing to decide on and select EC2 instance types. With EMR Serverless, you can create an application that runs either Spark or Hive as the processing engine, and then select a maximum for CPU, memory, and storage for that application. You can optionally select to pre-initialize Spark drivers and workers for your application, which then creates a

warm pool of workers for an application. While there is a cost to have the warm pool running, it does enable submitted jobs to start processing immediately, and you can optionally specify a timeout, after which the application will stop if idle. If you do not pre-initialize capacity, it may take a few minutes for a submitted job to run, but it is the most cost-effective way to run your jobs as you do not pay for application idle time. Once an application is running, you can submit multiple Spark jobs to the application.

As discussed above, Amazon EMR can be used to run Apache Spark, and you might be wondering why AWS has multiple services (Glue and EMR) that effectively offer the same big data processing engine. While either service can be used to perform big data processing using the Apache Spark engine, there are differences from a management, integration, and cost perspective.

AWS Glue requires the least amount of configuration, as you just need to specify a worker type and a maximum number of DPUs for each job and can then submit a Spark job to run. Using EMR Serverless requires a bit more configuration as you need to configure an application, and can then submit jobs to that application. However, you have more control over both the number of CPUs and the memory for each worker, and the cost for equivalent processing power is slightly lower than Glue.

Amazon EMR provisioned clusters (on EC2 or EKS) require more expertise to manage but provide the most control for fine-tuning your Spark jobs. They also support a wide range of big data processing applications beyond Spark (such as Hive, Presto, and Trino) and are the lowest-cost method for running Spark jobs using a managed AWS service.

One of the other differences is that AWS Glue has a number of built-in connectors, as well as a marketplace with additional connectors, that can make it easier to integrate your Spark code with external systems.

The following diagram shows an EMR cluster provisioned on EC2, including some of the open-source projects that can be run on the cluster:

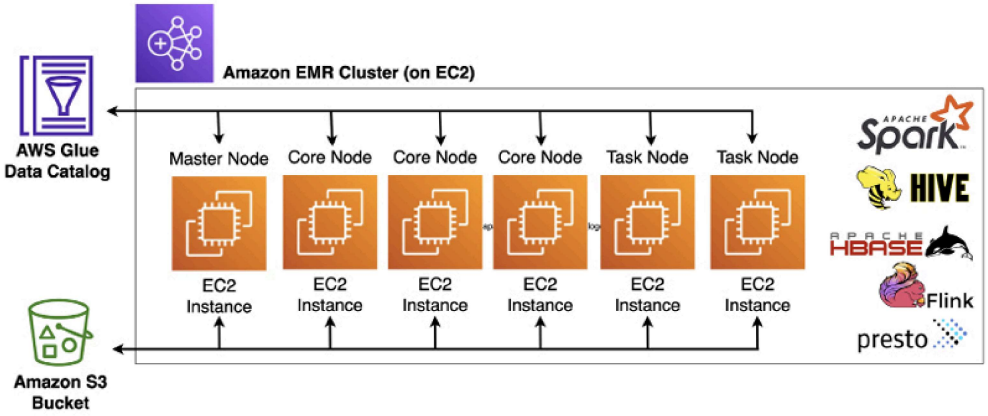

Figure 3.4: High-level overview of an EMR cluster running on EC2

Each EMR provisioned cluster requires a master node, at least one **core node** (a worker node that includes local storage), and then optionally a number of **task nodes** (worker nodes that do not have any local storage).

Having looked at how we can transform data in our data lake, let's now look at the AWS services that enable us to orchestrate the different components of our data pipeline.

An overview of AWS services for orchestrating big data pipelines

As discussed in *Chapter 2, Data Management Architectures for Analytics*, a data pipeline can be built to bring in data from source systems, and then transform that data, often moving the data through multiple stages, further transforming or enriching the data as it moves through each stage.

An organization will often have tens or hundreds of pipelines that work independently or in conjunction with each other on different datasets and perform different types of transformations. Each pipeline may use multiple services to achieve the goals of the pipeline, and orchestrating all the varying services and pipelines can be complex. In this section, we will look at a number of AWS services that help with this **orchestration** task.

AWS Glue workflows for orchestrating Glue components

In the *An overview of AWS services for transforming data* section, we covered AWS Glue, a service that includes a number of components. As a reminder, they are as follows:

- A serverless Apache Spark or Python shell environment for performing ETL transformations
- Glue Data Catalog, which provides a centralized logical representation (database and tables) of the physical data stored in Amazon S3
- Glue crawlers, which can be configured to examine files in a specific location, automatically infer the schema of the file, and add the file to the AWS Glue Data Catalog

AWS Glue workflows are a functionality within the AWS Glue service that has been designed to help orchestrate the various AWS Glue components. A workflow consists of an ordered sequence of steps that can run Glue crawlers and Glue ETL jobs (Spark or Python shell).

The following diagram shows a visual representation of a simple Glue workflow that can be built in the AWS Glue console:

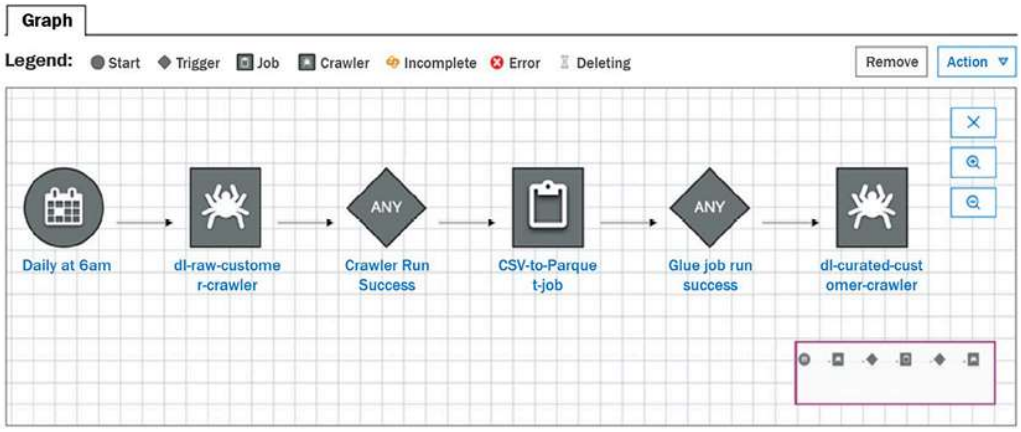

Figure 3.5: AWS Glue workflow

This workflow orchestrates the following tasks:

1. It runs a *Glue Crawler* to add newly ingested data from the raw zone of the data lake into Glue Data Catalog.
2. Once the Glue Crawler completes, it triggers a *Glue ETL job* to convert the raw CSV data into Parquet format and writes it to the curated zone of the data lake.

3. When the Glue job is complete, it triggers a Glue Crawler to add the newly transformed data to the curated zone in Glue Data Catalog.

Each step of the workflow can retrieve and update the state information about the workflow. This enables one step of a workflow to provide state information that can be used by a subsequent step in the workflow. For example, a workflow may run multiple ETL jobs, and each ETL job can update state information, such as the location of files that it outputted, which will be available to be used by subsequent workflow steps.

The preceding diagram is an example of a relatively simple workflow, but AWS Glue workflows are capable of orchestrating much more complex workflows. However, it is important to note that Glue workflows can only be used to orchestrate Glue components, which are ETL jobs and Glue crawlers.

If you only use AWS Glue components in your pipeline, then AWS Glue workflows are well suited to orchestrate your data transformation pipelines. But if you have a use case that needs to incorporate other AWS services in your pipeline (such as AWS Lambda), then keep reading to learn about other available options.

AWS Step Functions for complex workflows

Another option for orchestrating your data transformation pipelines is **AWS Step Functions**, a service that enables you to create complex workflows that can be integrated with over 220 AWS services, without needing to maintain code.

Step Functions is serverless, meaning that you do not need to deploy or manage any infrastructure, and you pay for the service based on your usage, not on fixed infrastructure costs.

With Step Functions, you use JSON to define a state machine using a structured language known as the **Amazon States Language** (**ASL**). Alternatively, you can use **Step Functions Workflow Studio** to create a workflow using a visual interface that supports dragging and dropping, and this creates the JSON ASL for you. The resulting workflow can run multiple tasks, run different branches based on a choice, enter a wait

state where you specify a delay before the next step is run, and loop back to previous steps, as well as various other things that can be done to control the workflow.

When you start a state machine, you include JSON data as input text that will be passed to the first state in the workflow. The first state in the workflow uses the input data, performs the function it is configured to do (such as running a Lambda function using the input passed into the state machine), modifies the JSON data, and then passes the modified JSON data to the next state in the workflow.

You can trigger a step function using **Amazon EventBridge** (such as on a schedule or in response to something else triggering an EventBridge event), as well as various other AWS services (such as Amazon API Gateway, AWS CodePipeline, or AWS IoT Rules Engine). You can also trigger a step function on-demand, by calling the Step Functions API.

The following is an example of a Step Functions state machine:

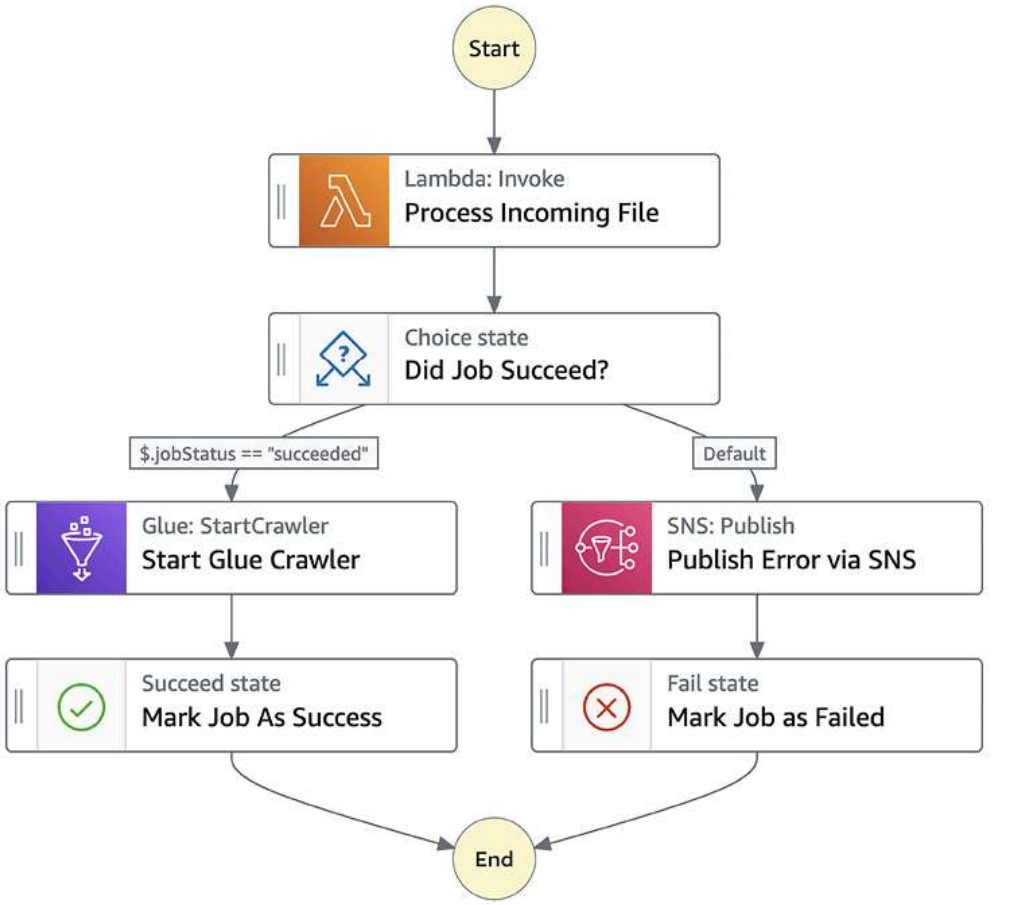

Figure 3.6: AWS Step Functions state machine

This state machine performs the following steps:

1. An EventBridge rule is triggered whenever a file is uploaded to a particular Amazon S3 bucket, and the EventBridge rule starts our state machine, passing in a JSON object that includes the location of the newly uploaded file.
2. The first step, **Process Incoming File**, runs a Lambda function that takes the location of the uploaded file as input and processes the incoming file (for example, converting from CSV into Parquet format). The output of the Lambda function indicates whether the file processing succeeded or failed. This information is included in the JSON passed to the next step.
3. The **Did Job Succeed?** step is of type `Choice`. It examines the JSON data passed to the step, and if the `jobStatus` field is set to `succeeded`, it branches to **Start Glue Crawler**. Otherwise, it branches to **Publish Error via SNS**.
4. In the **Start Glue Crawler** step, an AWS Glue Crawler is triggered. When this completes, a Step Functions flow runs that marks this step function execution as being successful.
5. In the **Publish Error via SNS** step, a message is published to an SNS topic, which sends an email or text message to an operator to inform them that the job failed. When this completes, a Step Functions flow runs that marks this step function execution as having failed.

In *Chapter 10, Orchestrating the Data Pipeline*, we have a hands-on exercise for orchestrating a pipeline using AWS Step Functions.

Amazon Managed Workflows for Apache Airflow (MWAA)

Apache Airflow is a popular open-source solution for orchestrating complex data engineering workflows. It was created by **Airbnb** in 2014 to help its internal teams manage its increasingly complex workflows and became a top-level Apache project in 2019.

Airflow enables users to create processing pipelines programmatically (using the Python programming language) and provides a user interface

to monitor the execution of the workflows. Complex workflows can be created and Airflow includes support for a wide variety of integrations, including integrations with services from AWS, Microsoft Azure, Google Cloud Platform, and others.

However, installing and configuring Apache Airflow in a way that can support the resilience and scaling required for large production environments is complex, and maintaining and updating the environment can be challenging. As a result, AWS created **Managed Workflows for Apache Airflow** (**MWAA**), which enables users to easily deploy a managed version of Apache Airflow that can automatically scale out additional workers as demand on the environment increases and scale in the number of workers as demand decreases.

An MWAA environment consists of the following components:

1. **Scheduler**: The scheduler runs a multithreaded Python process that controls what tasks need to be run, and where and when to run those tasks.
2. **Worker/executor**: The worker(s) execute(s) tasks. Each MWAA environment contains at least one worker, but when configuring the environment, you can specify the maximum number of additional workers that should be made available. MWAA automatically scales out the number of workers up to that maximum, but will also automatically reduce the number of workers as tasks are completed and if no new tasks need to run. The workers are linked to your VPC (the private network in your AWS account).
3. **Meta-database**: This runs in the MWAA service account and is used to track the status of tasks.
4. **Web server**: The web server also runs in the MWAA service account and provides a web-based interface that users can use to monitor and execute tasks.

Note that even though the meta-database and web server run in the MWAA service account, there are separate instances of these for every MWAA environment, and there are no components of the architecture that are shared between different MWAA environments.

When migrating from an on-premises environment where you already run Apache Airflow, or if your team already has Apache Airflow skills, then the MWAA service should be considered for managing your data processing pipelines and workflows in AWS. However, it is important to note that while this is a managed service (meaning that AWS deploys the environment for you and upgrades the Apache Airflow software), it is not a serverless environment.

With MWAA, you select a core environment size (small, medium, or large), and are charged based on the environment size, plus a charge for the amount of storage used by the meta-database and for any additional workers you make use of. Whether you run one 5-minute job per day or run multiple simultaneous jobs 24 hours a day 7 days a week, the charge for your core environment will remain the same. With serverless environments such as Amazon Step Functions, billing is based on a consumption model, so there is no underlying fixed charge.

Having looked at the services that can be used to orchestrate our data pipelines, let's now move on to an overview of the AWS services that can be used to query and analyze our data.

An overview of AWS services for consuming data

Once the data has been transformed and optimized for analytics, the various data consumers in an organization need easy access to the data via a number of different types of interfaces. Data scientists may want to use standard SQL queries to query the data, while data analysts may want to both query the data in place using SQL and also load subsets of the data into a high-performance data warehouse for low-latency, high-concurrency queries, and scheduled reporting. Business users may prefer to access data via a visualization tool that enables them to view data represented as graphs, charts, and other types of visuals.

In this section, we will introduce a number of AWS services that enable different types of data consumers to work with our optimized datasets. We won't cover all the services that can be used to consume data in this section, but instead will highlight the primary services relevant to a data engineering role.

Amazon Athena for SQL queries in the data lake

Amazon Athena is a serverless solution for using standard SQL queries to query data that exists in a data lake or in other data sources. As soon as a dataset has been written to Amazon S3 and cataloged in AWS Glue Data Catalog, users can run complex SQL queries against the data without needing to set up or manage any infrastructure.

What is SQL?

SQL is a standard language used to query relational datasets. A person proficient in SQL can draw information out of very large relational datasets easily and quickly, combining different tables, filtering results, and performing aggregations.

Data scientists and data analysts frequently use SQL to explore and better understand datasets that may be useful to them. Enabling these data consumers to query the data in an Amazon S3 data lake, without needing to first load the data into a traditional database system, increases productivity and flexibility for these data consumers.

Many tools are designed to interface with data via SQL and these tools often connect to the SQL data source using either a **JDBC** or **ODBC** database connection. Amazon Athena enables a data consumer to query datasets in the data lake (or other connected data sources) through the AWS Management Console interface or through a JDBC or ODBC driver.

Graphical SQL query tools, such as **SQL Workbench**, can connect to Amazon Athena via the JDBC driver, and you can programmatically connect to Amazon Athena and run SQL queries in your code through the ODBC driver.

Athena Federated Query, a feature of Athena, enables you to build connectors so that Athena can query other data sources, beyond just the data in an S3 data lake. Amazon provides a number of pre-built open-source connectors for Athena, enabling you to connect Athena to sources such as **Amazon DynamoDB** (a NoSQL database), as well as other Amazon-managed relational database engines, and even Amazon CloudWatch Logs, a

centralized logging service. Using this functionality, a data consumer can run a query using Athena that gets active orders from Amazon DynamoDB, references that data against the customer database running on PostgreSQL, and then brings in historical order data for that customer from the S3 data lake – all in a single SQL statement.

At re:Invent 2022, AWS announced **Amazon Athena for Apache Spark**, a new functionality that enables running Apache Spark jobs with Athena. With this new feature, you can launch a Jupyter notebook from within the Athena console, and query data in the data lake using Apache Spark code.

Amazon Redshift and Redshift Spectrum for data warehousing and data lakehouse architectures

Data warehousing is not a new concept or technology (as we discussed in *Chapter 2, Data Management Architectures for Analytics*), but **Amazon Redshift** was the first cloud-based data warehouse to be created. Launched in 2012, it was AWS's fastest-growing service by 2015, and today there are tens of thousands of customers that use it.

A Redshift data warehouse is designed for reporting and analytic workloads, commonly referred to as **Online Analytical Processing** (**OLAP**) workloads. Redshift provides a clustered environment that enables all the compute nodes in a cluster to work with portions of the data involved in a SQL query, helping to provide the best performance for scenarios where you are working with data that has been stored in a highly structured manner, and you need to do complex joins across multiple large tables on a regular basis. As a result, Redshift is an ideal query engine for reporting and visualization services that need to work with large datasets.

A typical SQL query that runs against a Redshift cluster would be likely to retrieve data from hundreds of thousands, or even millions, of rows in the database, often performing complex joins between different tables, and likely doing calculations such as aggregating, or averaging, certain columns of data. The queries run against the data warehouse will often be used to answer questions such as "*What was the average sale amount for sales in our stores last month, broken down by each ZIP code of the*

USA?" or "*Which products, across all of our stores, have seen a 20% increase in sales between Q4 of last year and Q1 of this year?*"

In a modern analytic environment, a common use case for a data warehouse would be to load a subset of data from the data lake into the warehouse, based on which data needs to be queried most frequently and which data needs to be used for queries requiring the best possible performance.

In this scenario, a data engineer may create a pipeline to load customer, product, sales, and inventory data into the data warehouse on a daily basis. Knowing that 80% of the reporting and queries will be on the last 12 months of sales data, the data engineer may also design a process to remove all data that's more than 12 months old from the data warehouse.

But what about the 20% of queries that need to include historical data that's more than 12 months old? That's where Redshift Spectrum comes in, a feature of Amazon Redshift that enables a user to write a single query that queries data that has been loaded into the data warehouse, as well as data that exists outside the data warehouse, in the data lake. To enable this, the data engineer can configure the Redshift cluster to connect with AWS Glue Data Catalog, where all the databases and tables for our data lake are defined. Once that has been configured, a user can reference both internal Redshift tables and tables registered in Glue Data Catalog.

The following diagram shows the Redshift and Redshift Spectrum architecture:

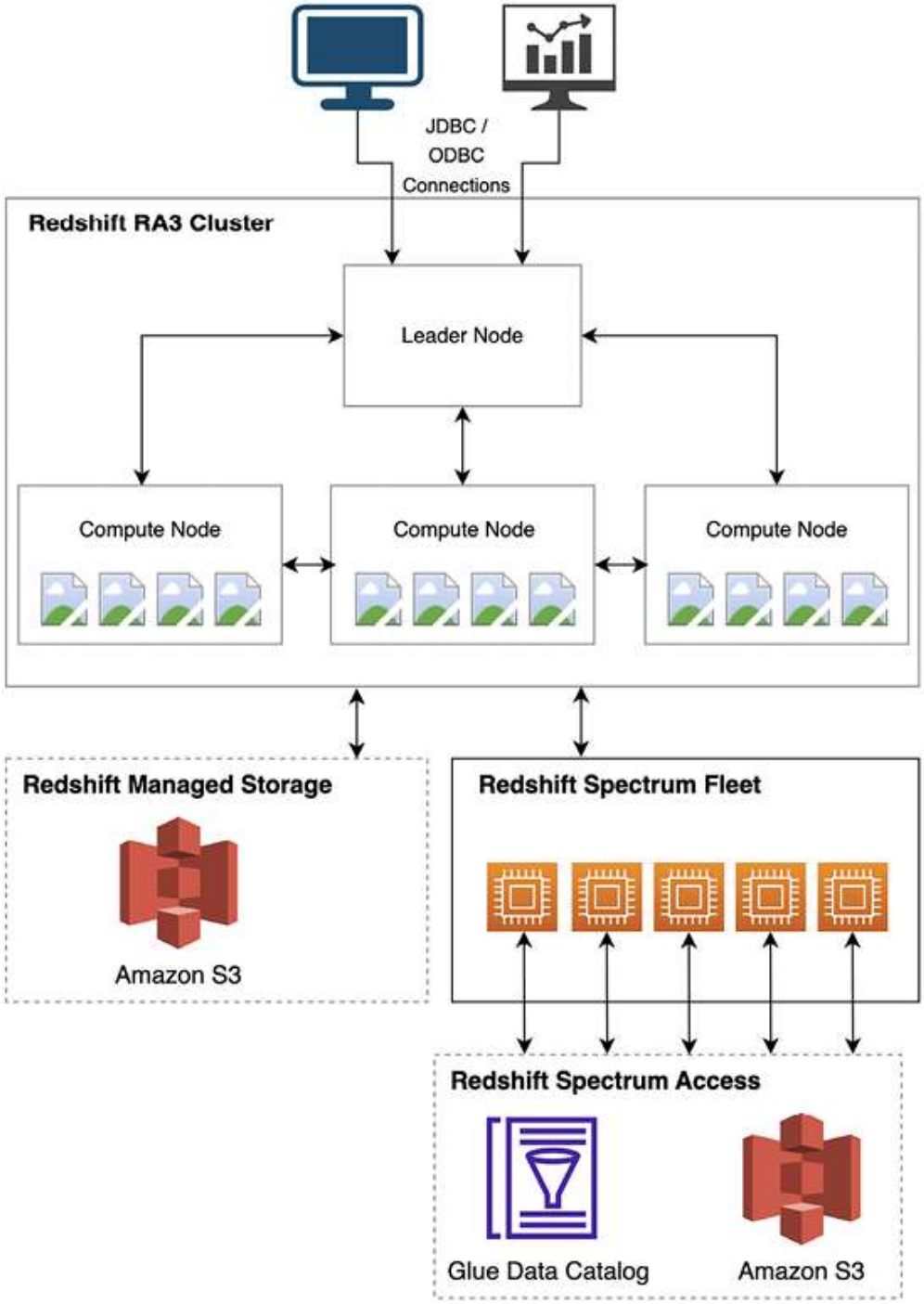

Figure 3.7: Redshift architecture

In the preceding diagram, we can see that a user connects to the Redshift leader node (via JDBC or ODBC). This node does not query data directly but is effectively the *central* brain behind all the queries that run on the cluster. In a scenario where a user is running a query that needs to query both current (last 12 months of) sales data and historical sales data in Amazon S3, the process works as follows:

1. Using a SQL client, the user makes a connection, authenticates with the Redshift leader node, and sends through a SQL statement that queries both the `current_sales` table (a table in which the data exists either in the local SSD cache of the cluster, or within **Redshift Managed Storage** (**RMS**), and contains the past 12 months of sales data) and the `historical_sales` table (a table that is registered in Glue Data Catalog, and where the data files are located in the Amazon S3 data lake, which contains historical sales data going back 10 years).
2. The leader node analyzes and optimizes the query, compiles a query plan, and pushes individual query execution plans to the compute nodes in the cluster.
3. The compute nodes query data they have locally (for the `current_sales` table) and query AWS Glue Data Catalog to gather information on the external `historical_sales` table. Using the information they gather, they can optimize queries for the external data and push those queries out to the Redshift Spectrum layer.
4. Redshift Spectrum is outside of a customer's Redshift cluster and is made up of thousands of worker nodes (Amazon EC2 compute instances) in each AWS Region. These worker nodes are able to scan, filter, and aggregate data from the files in Amazon S3 and then stream the results back to the Amazon Redshift cluster.
5. The Redshift cluster performs final operations to join and merge data and then returns the results to the user's SQL client.
6. Note the difference between **RMS** and data in S3 that Redshift Spectrum is able to query. Files that Redshift stores in RMS are managed by Redshift and are not accessible outside of Redshift. A Redshift cluster will automatically move data between the S3-based RMS and local SSD storage in each node of the cluster and is designed to ensure that frequently queried data is available from the local SSD drives for best performance. With Redshift Spectrum, any data in Amazon S3 that has been cataloged in Glue Data Catolog can be queried by a fleet of compute instances that are available to all Redshift clusters. This could be data in a variety of formats (such as JSON, CSV, Parquet, or others) and can be queried by other tools, such as Amazon Athena, since this data is not managed by Redshift.
7. Redshift also includes a number of advanced features, such as data sharing between Redshift clusters, automatic optimization of tables, integration with machine learning through Redshift ML, and the abil-

ity to mask data through data masking policies. We will do a deeper dive into Amazon Redshift in *Chapter 9*, *Loading Data into a Data Mart*, and this includes a hands-on exercise for loading data into Redshift and then querying that data.

Overview of Amazon QuickSight for visualizing data

"*A picture is worth a thousand words*" is a common saying, and it is a sentiment that most business users would strongly agree with. Imagine for a moment that you are a busy sales manager, it's Monday morning, and you need to quickly determine how your various sales territories performed last quarter before your 9 a.m. call.

Your one option is to receive a detailed spreadsheet showing the specific sales figures broken down by territory and segment, as per *Figure 3.8*:

Sales Data by Territory and Segment			
Territory	**SMB**	**Midmarket**	**Enterprise**
East Q3	$ 168,778	$ 210,696	$ 423,875
East Q4	$ 196,254	$ 244,995	$ 492,878
South Q3	$ 99,361	$ 168,572	$ 263,119
South Q4	$ 116,895	$ 198,320	$ 309,552
Central Q3	$ 132,882	$ 203,082	$ 296,332
Central Q4	$ 156,332	$ 238,920	$ 245,000
Mountain Q3	$ 127,699	$ 213,247	$ 271,440
Mountain Q4	$ 146,780	$ 245,112	$ 312,000
West Q3	$ 156,147	$ 210,558	$ 396,885
West Q4	$ 185,889	$ 250,664	$ 526,995

Figure 3.8: Sales table showing sales data by territory and segment

The other option you have is to receive a graphical representation of the data in the form of a bar graph, as shown in *Figure 3.9*. Within the interface, you can filter data by territory and market segment, and also drill down to get more detailed information:

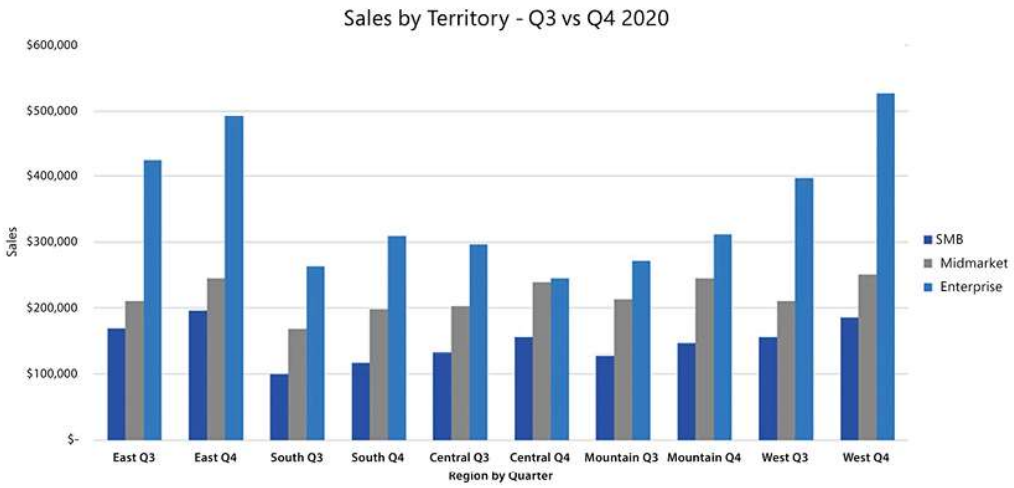

Figure 3.9: Sample graph showing sales data by territory and segment

Most people would prefer the graphical representation of the data, as they can easily visually compare sales between quarters, segments, and territories, or identify the top sales territory and segment with just a glance. As a result, the use of business intelligence tools, which provide visual representations of complex data, is extremely popular in the business world.

Amazon QuickSight is a service from AWS that enables the creation of these types of complex visualizations, but beyond just providing static visuals, the charts created by QuickSight enable users to filter data and drill down to get further details. For example, our sales manager could filter the visual to just see the numbers from Q4 or to just see the enterprise segment. The user could also drill down into the Q4 data for the enterprise segment in the West territory to see sales by month, for example.

Amazon QuickSight is serverless, which means there are no servers for the organization to set up or manage, and there is a simple monthly fee based on the user type (either an author, who can create new visuals, or a reader, who can view visuals created by authors).

A data engineer can configure QuickSight to access data from a multitude of sources, including accessing data in an Amazon S3-based data lake via integration with Amazon Athena. In *Chapter 12*, *Visualizing Data with Amazon QuickSight*, we will do a hands-on exercise to create a simple visualization with QuickSight.

In the next section, we will wrap up the chapter by getting hands-on with building a simple transformation that converts a CSV file into Parquet format, using Lambda to perform the transformation.

Hands-on – triggering an AWS Lambda function when a new file arrives in an S3 bucket

In the hands-on portion of this chapter, we're going to configure an S3 bucket to automatically trigger a Lambda function whenever a new file is written to the bucket. In the Lambda function, we're going to make use of an open-source Python library called **AWS SDK for pandas**, created by AWS Professional Services to simplify common ETL tasks when working in an AWS environment. We'll use the AWS SDK for pandas library to convert a CSV file into Parquet format and then update AWS Glue Data Catalog.

Converting a file into Parquet format is a common transformation in order to improve analytic query performance against our data. This can either be done in bulk (such as by using an AWS Glue job that runs every hour to convert files received in the past hour), or it can be done as each file arrives, as we are doing in this hands-on exercise. A similar approach can be used for other use cases, such as updating a `total_sales` value in a database as files are received with daily sales figures from a company's retail stores across the world.

Let's get started with the hands-on section of this chapter.

Creating a Lambda layer containing the AWS SDK for pandas library

Lambda layers allow your Lambda function to bring in additional code, packaged as a **.zip** file. In our use case, the Lambda layer is going to contain the AWS SDK for pandas Python library, which we can then attach to any Lambda function where we want to use the library.

To create a Lambda layer, do the following:

1. Access the 2.19.0 version of the AWS SDK for pandas library in GitHub at `https://github.com/aws/aws-sdk-pandas/releases`. Under **Assets**, download the `awswrangler-layer-2.19.0-py3.9.zip` file to your local drive. Note that there is a direct link to this file on the GitHub site for this book at `https://github.com/PacktPublishing/Data-Engineering-with-AWS-2nd-edition/tree/main/Chapter03`.
2. Log in to the **AWS Management Console** as the administrative user you created in *Chapter 1, An Introduction to Data Engineering* (`https://console.aws.amazon.com`).
3. Make sure that you are in the Region that you have chosen for performing the hands-on sections in this book. The examples in this book use the `us-east-2` (**Ohio**) Region.
4. In the top search bar of the AWS console, search for and select the **Lambda** service.
5. In the left-hand menu, under **Additional Resources**, select **Layers**, and then click on **Create layer**.
6. Provide a **name** for the layer (for example, `awsSDKpandas219_python39`) and an optional description, and then upload the `.zip` file you downloaded from GitHub. For **Compatible runtimes-optional**, select **Python 3.9**, and then click **Create**. The following screenshot shows the configuration for this step:

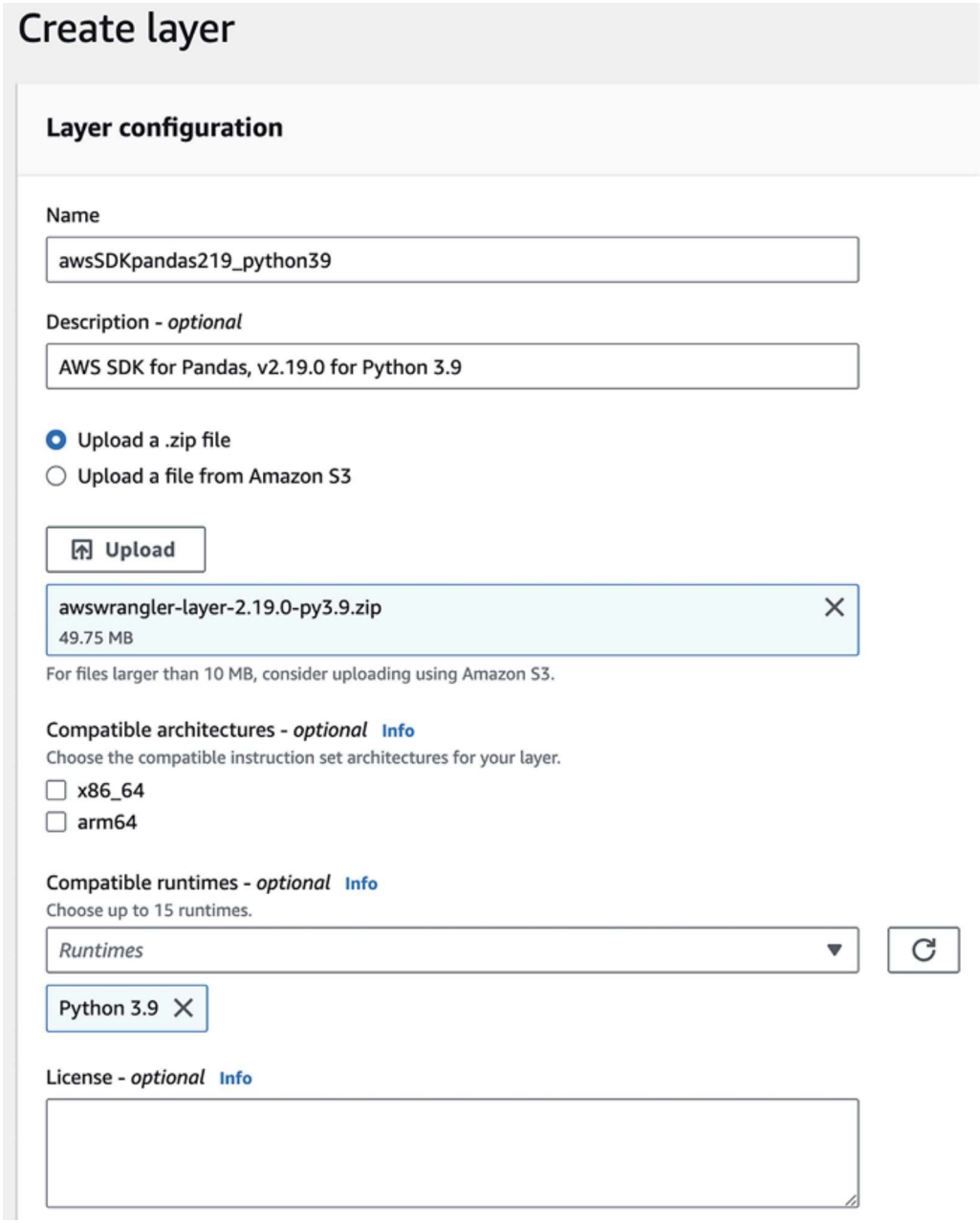

Figure 3.10: Creating and configuring an AWS Lambda layer

By creating a Lambda layer for the AWS SDK for pandas library, we can use AWS SDK for pandas in any of our Lambda functions just by ensuring this Lambda layer is attached to the function.

Creating an IAM policy and role for your Lambda function

In the previous chapter, we created three Amazon S3 buckets – one for a landing zone (for ingestion of raw files), one for a clean zone (for files

that have undergone initial processing and optimization), and one for the curated zone (to contain our finalized datasets, ready for consumption).

In this section, we will create a Lambda function to be triggered every time a new file is uploaded to our landing zone S3 bucket. The Lambda function will process the file and write out a new version of the file to a target bucket (our clean zone S3 bucket).

For this to work, we need to ensure that our Lambda function has the following permissions:

1. Read our source S3 bucket (for example, `dataeng-landing-zone-<initials>`).
2. Write to our target S3 bucket (for example, `dataeng-clean-zone-<initials>`).
3. Write logs to Amazon CloudWatch.
4. Access to all Glue API actions (to enable the creation of new databases and tables).

To create a new AWS IAM role with these permissions, follow these steps:

1. In the search bar at the top of the AWS console, search for and select the **IAM** service, and in the left-hand menu, select **Policies** and then click on **Create policy**.
2. By default, the **Visual editor** tab is selected, so click on **JSON** to change to the **JSON** tab.
3. Provide the JSON code from the following code blocks, replacing the boilerplate code. Note that you can also copy and paste this policy by accessing the policy on this book's GitHub page (**https://github.com/PacktPublishing/Data-Engineering-with-AWS-2nd-edition/blob/main/Chapter03/DataEngLambdaS3CWGluePolicy.json**). Note that if doing a copy and paste from the GitHub copy of this policy, you must replace `dataeng-landing-zone-<initials>` with the name of the landing zone bucket you created in *Chapter 2*, and replace `dataeng-clean-zone-<initials>` with the name of the clean zone bucket you created in *Chapter 2*.
   This first block of the policy configures the policy document and provides permissions for using CloudWatch log groups, log streams, and

log events:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
```

This next block of the policy provides permissions for all Amazon S3 actions (`get` and `put`) that are in the Amazon S3 bucket specified in the resource section (in this case, our clean zone and landing zone buckets). Make sure you replace `dataeng-clean-zone-<initials>` and `dataeng-landing-zone-<initials>` with the names of the S3 buckets you created in *Chapter 2*:

```
        {
            "Effect": "Allow",
            "Action": [
                "s3:*"
            ],
"Resource": [
                "arn:aws:s3:::dataeng-landing-zone-INITIALS/*",
                "arn:aws:s3:::dataeng-landing-zone-INITIALS",
                "arn:aws:s3:::dataeng-clean-zone-INITIALS/*",
                "arn:aws:s3:::dataeng-clean-zone-INITIALS"
            ]
        },
```

In the final statement of the policy, we provide permissions to use all AWS Glue actions (create job, start job, and delete job). Note that in a production environment, you should limit the scope specified in the resource section:

```
        {
            "Effect": "Allow",
            "Action": [
                "glue:*"
            ],
            "Resource": "*"
        }
    ]
}
```

4. Click on **Next Tags** and then **Next: Review**.
5. Provide a name for the policy, such as `DataEngLambdaS3CWGluePolicy`, and then click **Create policy**.
6. In the left-hand menu, click on **Roles** and then **Create role**.
7. For the trusted entity, ensure **AWS service** is selected, and for the service, select **Lambda** and then click **Next: Permissions**. In *step 4* of the next section (*Creating a Lambda function*), we will assign this role to our Lambda function.
8. Under **Attach permissions**, select the policy we just created (for example, `DataEngLambdaS3CWGluePolicy`) by searching and then clicking the tick box. Then, click **Next**.
9. Provide a role name, such as `DataEngLambdaS3CWGlueRole`, and click **Create role**.

Creating a Lambda function

We are now ready to create our Lambda function that will be triggered whenever a CSV file is uploaded to our source S3 bucket. The uploaded CSV file will be converted into Parquet, written out to the target bucket, and added to the Glue catalog using the AWS SDK for pandas library:

1. In the AWS console, search for and select the **Lambda** service, and in the left-hand menu, select **Functions** and then click **Create function**. *Make sure you are in the same Region as where you created your AWS buckets* in *Chapter 2*.
2. Select **Author from scratch** and provide a **function name** (such as `CSVtoParquetLambda`).
3. For **Runtime**, select **Python 3.9** from the drop-down list.

4. Expand **Change default execution role** and select **Use an existing role**. From the drop-down list, select the role you created in the previous section (such as `DataEngLambdaS3CWGlueRole`):

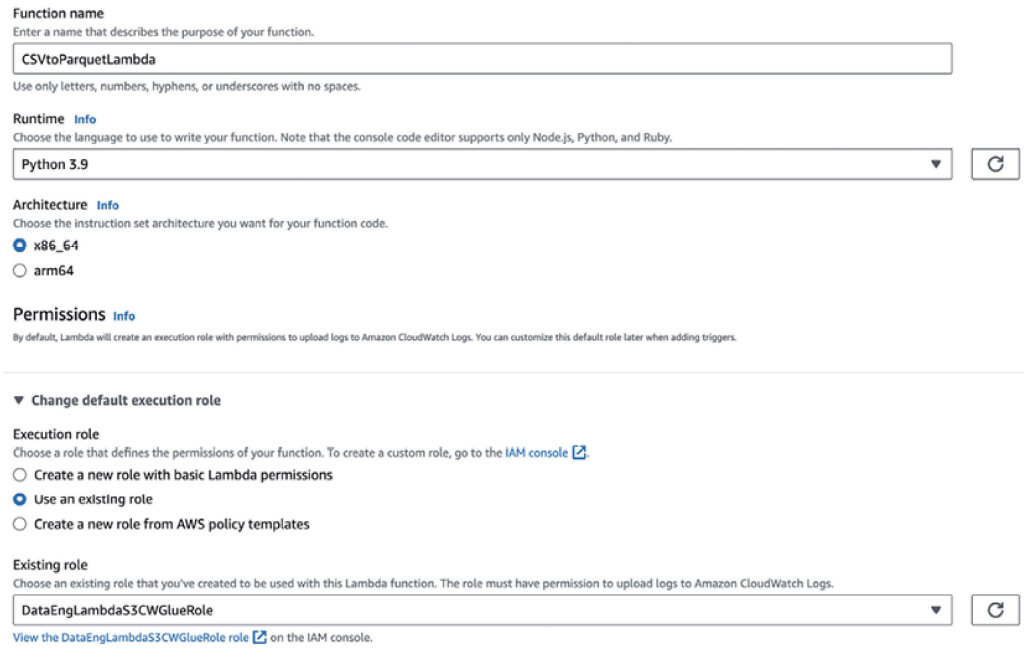

Figure 3.11: Creating and configuring a Lambda function

5. Do not change any of the **Advanced settings** and click **Create function**.
6. Click on **Layers** in the first box (*Function overview*), and then click **Add a layer** in the **Layers** box.
7. Select **Custom layers**, and from the dropdown, select the AWS SDK for pandas layer you created in a previous step (such as `awsSDKpandas219_python39`). Select the latest version and then click **Add**.

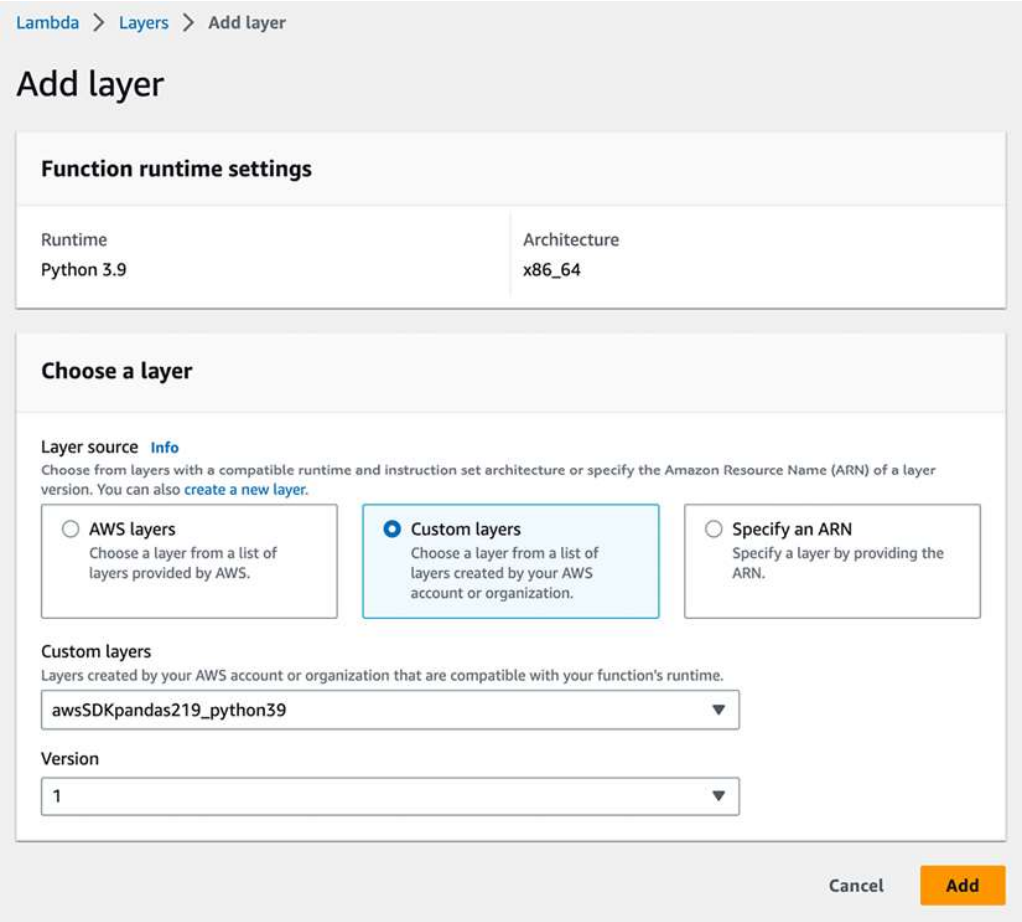

Figure 3.12: Adding an AWS Lambda layer to an AWS Lambda function

8. Scroll down to the **Code Source** section in the Lambda console. The following code can be downloaded from this book's GitHub repository. Make sure to replace any existing code in `lambda_function` with this code.
   In the first few lines of code, we import `boto3` (the AWS Python SDK), `awswrangler` (which is part of the AWS SDK for pandas library that we added as a Lambda layer), and a function from the `urllib` library called `unquote_plus`:

   ```
   import boto3
   import awswrangler as wr
   from urllib.parse import unquote_plus
   ```

   We then define our main function, `lambda_handler`, which is called when the Lambda function is executed. The `event` data contains information such as the S3 object that was uploaded and was the cause

of the trigger that ran this function. From this event data, we get the S3 bucket name and the object key. We also set the Glue catalog `db_name` and `table_name` based on the path of the object that was uploaded (review the comments in the code below for an explanation of how this works).

```
def lambda_handler(event, context):
    # Get the source bucket and object name as passed to the Lambda function
    for record in event['Records']:
        bucket = record['s3']['bucket']['name']
        key = unquote_plus(record['s3']['object']['key'])

    # We will set the DB and table name based on the last two elements of
    # the path prior to the filename. If key = 'dms/sakila/film/LOAD01.csv',
    # then the following lines will set db to 'sakila' and table_name to 'fil
    key_list = key.split("/")
    print(f'key_list: {key_list}')
    db_name = key_list[len(key_list)-3]
    table_name = key_list[len(key_list)-2]
```

We now print out some debugging information that will be captured in our Lambda function logs. This includes information such as the Amazon S3 bucket and key that we are processing. We then set the `output_path` value here, which is where we are going to write the Parquet file that this function creates. **Make sure to change the `output_path` value of this code to match the name of the target S3 bucket you created earlier**:

```
    print(f'Bucket: {bucket}')
    print(f'Key: {key}')
    print(f'DB Name: {db_name}')
    print(f'Table Name: {table_name}')

    input_path = f"s3://{bucket}/{key}"
    print(f'Input_Path: {input_path}')
    output_path = f"s3://dataeng-clean-zone-INITIALS/{db_name}/{table_name}"
    print(f'Output_Path: {output_path}')
```

We can then use the AWS SDK for pandas library (defined as `wr` in our function) to read the CSV file that we received. We read the con-

tents of the CSV file into a `pandas` DataFrame we are calling `input_df`. We also get a list of current Glue databases, and if the database we want to use does not exist, we create it:

```
    input_df = wr.s3.read_csv([input_path])

    current_databases = wr.catalog.databases()
    wr.catalog.databases()
    if db_name not in current_databases.values:
        print(f'- Database {db_name} does not exist ... creating')
        wr.catalog.create_database(db_name)
    else:
        print(f'- Database {db_name} already exists')
```

Finally, we can use the AWS SDK for pandas library to create a Parquet file containing the data we read from the CSV file. For the S3 to Parquet function, we specify the name of the DataFrame (`input_df`) that contains the data we want to write out in Parquet format. We also specify the S3 output path, the Glue database, and the table name:

```
    result = wr.s3.to_parquet(
        df=input_df,
        path=output_path,
        dataset=True,
        database=db_name,
        table=table_name,
        mode="append")
    print("RESULT: ")
    print(f'{result}')
    return result
```

9. Click on **Deploy** at the top of the **Code Source** window
10. Click on the **Configuration** tab (above the **Code Source** window), and on the left-hand side, click on **General configuration**. Click the **Edit** button and modify the **Timeout** to be 1 minute (the default timeout of 3 seconds is likely to be too low to convert some files from CSV into Parquet format). Then, click on **Save**. *If you skip this step, you are likely to get an error when your function runs.*

Configuring our Lambda function to be triggered by an S3 upload

Our final task is to configure the Lambda function so that whenever a CSV file is uploaded to our landing zone bucket, the Lambda function runs and converts the file into Parquet format:

1. In the **Function Overview** box of our Lambda function, click on **Add trigger**.
2. For **Trigger configuration**, select the **S3** service from the drop-down list.
3. For **Bucket**, select your landing zone bucket (for example, `dataeng-landing-zone-<initials>`).
4. We want our rule to trigger whenever a new file is created in this bucket, no matter what method is used to create it (**Put**, **Post**, or **Copy**), so select **All object create events** from the list.
5. For **suffix**, enter `.csv`. This will configure the trigger to only run the Lambda function when a file with a `.csv` extension is uploaded to our landing zone bucket.
6. Acknowledge the warning about **Recursive invocation**, which can crop up if you set up a trigger on a specific bucket to run a Lambda function and then you get your Lambda function to create a new file in that same bucket and path. This is a good time to double-check and make sure that you are configuring this trigger in the `LANDING ZONE` bucket (for example, `dataeng-landing-zone-<initials>`) and not the target `CLEAN ZONE` bucket that our Lambda function will write to:

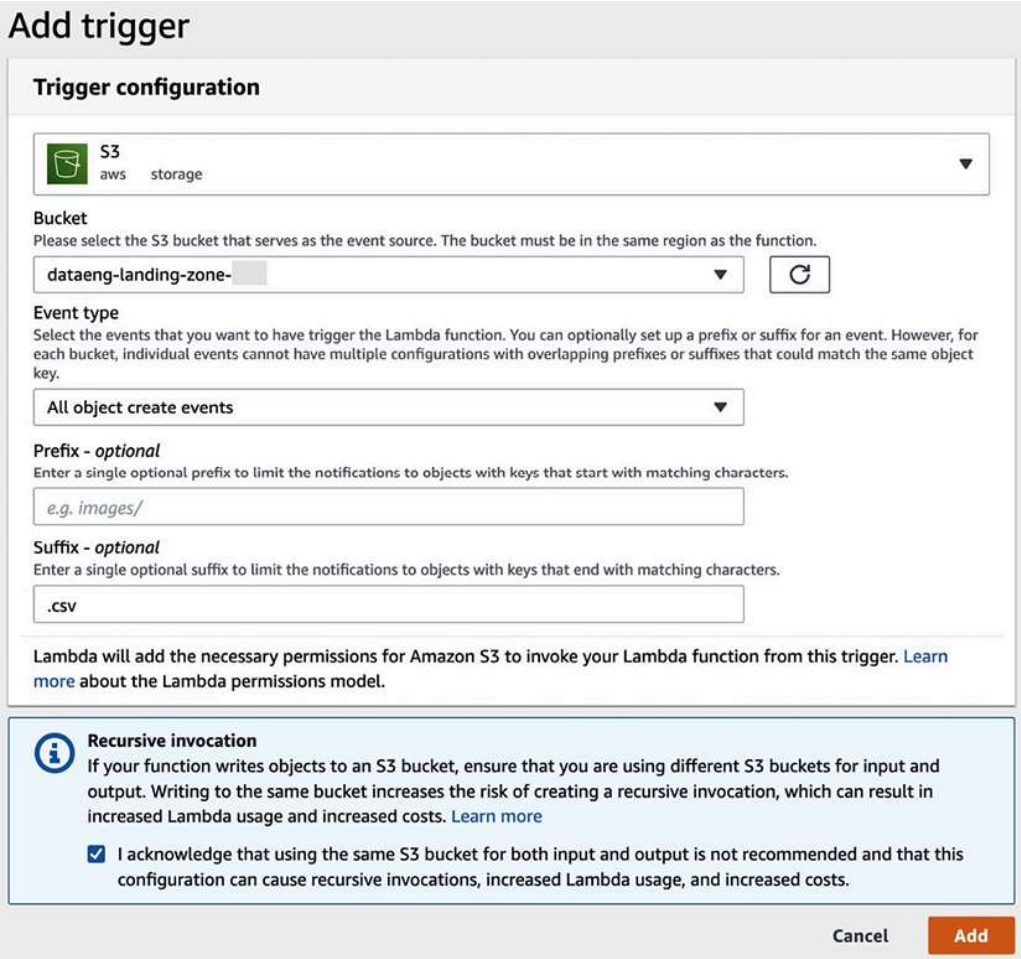

Figure 3.13: Configuring an S3-based trigger for an AWS Lambda function

7. Click **Add** to create the trigger.
8. Create a simple CSV file called `test.csv` that you can use to test the trigger or download `test.csv` from the GitHub site for this chapter (**https://github.com/PacktPublishing/Data-Engineering-with-AWS-2nd-edition/blob/main/Chapter03/test.csv**).
   Ensure that the first line has column headings, as per the following example:

```
name,favorite_num
Gareth,23
Tracy,28
Chris,16
Emma,14
```

   Ensure you create the file with a standard text editor, and not Word processing software (such as Microsoft Word) or any other software

that will add additional formatting to the file.

9. Navigate to the Amazon S3 console (https://s3.console.aws.amazon.com/s3) and click on the `dataeng-landing-zone-initials` bucket that you previously created. Then click on **Create folder** and provide a folder name of `cleanzonedb`. The top-level folder we create here is going to be used as the name of the database that will be created in Glue Data Catalog to store our new table.
10. Navigate into the `cleanzonedb` folder, and create a second-level folder (this will be used as the name of the table that gets created in Glue Data Catalog). Name the second-level folder `csvtoparquet`.
11. Navigate into the `csvtoparquet` folder, click on **Upload**, and then upload the `test.csv` file you previously created. The file should be uploaded into `dataeng-landing-zone-initials/cleanzonedb/csvtoparquet`.
12. If everything has been configured correctly, your Lambda function will have been triggered and will have written out a Parquet-formatted file to your target S3 bucket and created a Glue database and table. You can access the Glue service in the AWS Management Console to ensure that a new database and table have been created in the data catalog and can run the following command in CloudShell to ensure that a Parquet file has been written to your target bucket. Make sure to replace `dataeng-clean-zone-initials` with the name of your target S3 bucket:

```
aws s3 ls s3://dataeng-clean-zone-initials/cleanzonedb/csvtoparquet/
```

The result of this command should display the Parquet file that was created by the Lambda function.

Summary

In this chapter, we covered a lot! We reviewed a range of AWS services at a high level, including services for ingesting data from a variety of sources, services for transforming data, and services for consuming and working with data.

We then got hands-on, building a solution in our AWS account that converted a file from CSV format into Parquet format and registered the data in AWS Glue Data Catalog.

In the next chapter, we will cover a really important topic that all data engineers need to have a good understanding of and that needs to be central to every project that a data engineer works on, and that is the topic of data governance.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/9s5mHNyECd