6
Ingesting Batch and Streaming Data

Having developed a high-level architecture for our data pipeline, we can now dive deep into the varied components of the architecture. We will start with data ingestion so that in the hands-on section of this chapter, we can ingest data and use that data for the hands-on activities in future chapters.

Data engineers are often faced with the challenge of *the five Vs of data*. These are the **variety** of data (the diverse types and formats of data); the **volume** of data (the size of the dataset); the **velocity** of the data (how quickly the data is generated and needs to be ingested); the **veracity** or **validity** of the data (the quality, completeness, and credibility of the data); and finally, the **value** of data (the value that the data can provide the business with).

In this chapter, we will look at several different types of data sources and examine the various tools available within AWS for ingesting data from these sources. We will also look at how to decide between multiple different ingestion tools to ensure you are using the right tool for the job. In the hands-on portion of this chapter, we will ingest data from both streaming and batch sources.

In this chapter, we will cover the following topics:

- Understanding data sources
- Ingesting data from database sources
- Ingesting data from streaming sources
- Hands-on – ingesting data from a database source
- Hands-on – ingesting data from a streaming source

Technical requirements

In the hands-on sections of this chapter, we will use the AWS **Database Migration Service (DMS)** service to ingest data from a database source, and then we will ingest streaming data using Amazon Kinesis. To ingest data from a database, you need IAM permissions that allow your user to create an RDS database, an EC2 instance, a DMS instance, and a new IAM role and policy.

For the hands-on section on ingesting streaming data, you will need IAM permissions to create a Kinesis Data Firehose instance, as well as permissions to deploy a CloudFormation template. The CloudFormation template that is deployed will create IAM roles, a Lambda function, as well as Amazon Cognito users and other Cognito resources.

To query the newly ingested data, you will need permission to create an AWS Glue Crawler and permission to use Amazon Athena to query data.

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/Chapter06.

Understanding data sources

Over the past decade, the amount and the variety of data that gets generated each year has significantly increased. Today, industry analysts talk about the volume of global data generated in a year in terms of **zettabytes** (**ZB**), a unit of measurement equal to a billion **terabytes** (**TB**). By some estimates, a little over 1 ZB of data existed in the world in 2012, and yet by the end of 2025, there will be an estimated 181 ZB of data created, captured, copied, and consumed worldwide.

In our pipeline whiteboarding session (covered in *Chapter 5*, *Architecting Data Engineering Pipelines*), we identified several data sources that we wanted to ingest and transform to best enable our data consumers. For each data source that is identified in a whiteboarding session, you need to develop an understanding of the variety, volume, velocity, veracity, and value of data; we'll move on to cover those now.

Data variety

In the past decade, the variety of data used in data analytics projects has greatly increased. If all data was simply relational data in a database (and there was a time when nearly all data was like this), it would be relatively easy to transfer into data analytic solutions. But today, organizations find value, and often a competitive advantage, in being able to analyze many other types of data (web server log files, photos, videos, and other media, geolocation data, sensor data, other IoT data, and so on).

For each data source in a pipeline, data engineers need to determine what type of data will be ingested. Data is typically categorized as being of one of three types, as we examine in the following subsections.

Structured data

Structured data is data that has been organized according to a data model, and is represented as a series of rows and columns. Each row represents a record, with the columns making up the fields of each record.

Each column in a structured data file contains data of a specific type (such as strings or numbers), and every row has the same number and type of columns. The definition of which columns are contained in each record, and the data type for each column, is known as the data schema.

Common data sources that contain structured data include the following:

- **Relational Database Management Systems** (**RDBMSes**) such as MySQL, PostgreSQL, SQL Server, and Oracle
- Delimited files such as **Comma-Separated Value** (**CSV**) files or **Tab-Separated Value** (**TSV**) files
- Spreadsheets such as Microsoft Excel files in **xls** format

The data shown in the following table is an example of structured data. In this case, it is data on the calorie content of several foods from the USA MyPyramid Food Raw Data, available at https://catalog.data.gov/dataset/mypyramid-food-raw-data. This data extract has been sorted to show some of the foods with the highest calorie content in the dataset:

Food_Code	Display_Name	Portion_Display_Name	Total Calo
71411000	Potato skin with cheese & bacon	order (10 halves)	1667.4
24301010	Roasted duck	duck half	1283.52
21103120	Breaded fried steak (eat lean & fat)	large steak	1069.2
28141010	Fried chicken frozen meal	large meal (16 oz)	1024.92
27347100	Chicken or turkey pot pie	16-ounce pie (Hungry Man)	976.1
58200100	Wrap sandwich (meat, vegetables, rice)	wrap	818.37
21103120	Breaded fried steak (eat lean & fat)	medium steak	801.9
58106730	Meat & veggie pizza, thick crust	small pizza (8" across)	798.64
24401010	Roasted Cornish game hen	hen	792.54
58106530	Meat pizza, thick crust	small pizza (8" across)	785.4

Figure 6.1: An example of structured data

Structured data can be easily ingested into analytic systems, including Amazon S3-based data lakes, and data marts such as an Amazon Redshift data warehouse.

Semi-structured data

Semi-structured data shares many of the same attributes as structured data, but the structure, or schema, does not need to be strictly defined. Generally, semi-structured data contains internal tags that identify data elements, but each record may contain different elements or fields.

Some of the data types in the unstructured data may be of a strong type, such as an integer value, while other elements may contain items such as free-form text. Common semi-structured formats include JSON and XML file formats.

The data that follows is part of a semi-structured JSON formatted file for product inventory. In this example, we can see that we have two items – a set of batteries and a pair of jeans:

```
[{
          "sku": 10001,
          "name": "Duracell – Copper Top AA Alkaline Batteries - long lasting, all-purpose 16 Count",
          "price": 12.78,
          "category": [{
           "id": "5443",
               "name": "Home Goods"
          }
           ],
          "manufacturer": "Duracell",
          "model": "MN2400B4Z"
     },
     {
          "sku": 10002,
          "name": "Levi's Men's 505 Jeans Fit Pants",
          "type": "Clothing",
          "price": 39.99,
```

```
          "fit_type": [{
                    "id": 855,
                    "description": "Regular"
               },
               {
                    "id": 902,
                    "description": "Big and Tall"
               }
          ],
          "size": [{
                    "id": 101,
                     "waist": 32,
                    "length": 32
               },
               {
                    "id": 102,
                    "waist": 30,
                    "length": 32
               }
          ],
          "category": [{
               "id": 3821,
               "name": "Jeans"
          }, {
               "id": 6298,
               "name": "Men's Fashion"
          }],
          "manufacturer": "Levi",
          "model": "00505-4891"
     }
]
```


While most of the fields are common between the two items, we can see that the pair of jeans includes attributes for `fit_type` and `size`, which are not relevant to batteries. You will also notice that the first item (the batteries) only belongs to a single category, while the jeans are listed in two categories (`Jeans` and `Men's Fashion`).

Capturing the same information in a structured data type, such as CSV, would be much more complex. CSV is not well suited to a scenario where different records have a different number of categories, for example, or where some records have additional attributes (such as `fit_type` or `size`). JSON is structured in a hierarchical format (where data can be nested, such as for `category`, `fit_type`, and `size`) and this provides significant flexibility.

Storing data in a semi-structured format, such as JSON, is commonly used for a variety of different use cases, such as working with IoT data, as well as for web and mobile applications.

Unstructured data

As a category, unstructured data covers many different types of data where the data does not have a predefined structure or schema. This can range from free-form data (such as text data in a PDF or word processing file or emails) to media files (such as photos, videos, satellite images, or audio files).

Some unstructured data can be analyzed directly, although generally not very efficiently unless using specialized tools. For example, it is generally not efficient to search large quantities of free-form text in a traditional database, although there are specialized tools that can be used for this purpose (such as **Amazon OpenSearch Service**).

However, some types of unstructured data cannot be directly analyzed with data analytic tools at all. For example, data analytic tools are unable to directly analyze image or video files. This does not mean that we cannot use these types of files in our analytic projects, but to make them useful for analytics, we need to process them further to extract useful metadata from the files.

A large percentage of the data that's generated today is considered unstructured data, and in the past few years, a lot of effort has been put into being able to make better use of this type of data. For example, we can use image recognition tools to extract metadata from an image or video file that can then be used in analytics. Or, we can use natural language processing tools to analyze free-form text reviews on a website to determine customer sentiment for different products.

Refer to *Chapter 13*, *Enabling Artificial Intelligence and Machine Learning*, for an example of how Amazon Comprehend can be used to extract sentiment analysis from product reviews.

Data volume

The next attribute of data that we need to understand for each of our data sources relates to the volume of data. For this, we need to understand both the total size of the existing data that needs to be ingested, as well as the daily/monthly/yearly growth of data.

For example, we may want to ingest data from a database that includes a one-time ingestion of 10 years of historical data totaling 2.2 TB in size. We may also find that data from this source generates around 30 GB of new data per month (or an average of 1 GB per day of new data). Depending on the network connection between the source system and the AWS target system, it may be possible to transfer the historical data over the network, but if we have limited bandwidth, we may want to consider using one of the devices in the *Amazon Snow* family of devices.

For example, we could load data onto an Amazon Snowball device that is shipped to our data center, and then send the device back to AWS, where AWS will load the data into S3 for us.

Understanding the volume of historical and future data also assists us in doing the initial sizing of AWS services for our use case, and helps us budget better for the associated costs.

Data velocity

Data velocity describes the speed at which we expect to ingest and process new data from the source system into our target system.

For ingestion, some data may be loaded according to a batch schedule once a day, or a few times a day (such as receiving data from a partner on a scheduled basis). Meanwhile, other

data may be streamed from the source to the target continually (such as when ingesting IoT data from thousands of sensors).

As an example, according to a case study on the AWS website, the BMW Group uses AWS services to ingest data from millions of BMW and MINI vehicles, processing TB of anonymous telemetry data every day. To read more about this, refer to the AWS case study titled *BMW Group Uses AWS-Based Data Lake to Unlock the Power of Data* (https://aws.amazon.com/solutions/case-studies/bmw-group-case-study/).

We need to have a good understanding of both how quickly our source data is generated (on a schedule or via real-time streaming data), as well as how quickly we need to process the incoming data (does the business only need insights from the data 24 hours after it is ingested, or is the business looking to gather insights in near real time?).

The velocity of data affects both how we ingest the data (such as through a streaming service such as Amazon Kinesis), as well as how we process the data (such as whether we run scheduled daily Glue jobs, or use Glue Streaming to continually process incoming data).

Data veracity

Data veracity considers various aspects of the data we're ingesting, such as the quality, completeness, and accuracy of the data.

As we discussed previously, the data we ingest may have come from a variety of sources, and depending on how the data was generated, the data may be incomplete or inconsistent. For example, data from IoT sensors where the sensor went offline for a while may be missing a period of data, or data captured from user forms or spreadsheets may contain errors or missing values.

We need to be aware of the veracity of the data we ingest so that we can ensure we take these items into account when processing the data. For example, some tools can help backfill missing data with averages, while others can help detect and remediate fields that contain invalid data.

Data value

Ultimately, all the data ingested and processed has been for a single purpose – finding ways to provide new insights and value to the business. While this is the last of the five Vs that we will discuss, it is the most important one to keep in mind when thinking of the bigger picture of what we are doing with data ingestion and processing.

We could ingest TB of data and clean and process the data in multiple ways, but if the end data product hasn't brought value to the business, then we have wasted both time and money.

When considering the data we are ingesting, we need to ensure we keep the big picture in mind. We need to make sure that it is worth ingesting this data and also understand how this data may add value to the business, either now or in the future.

Questions to ask

In *Chapter 5*, *Architecting Data Engineering Pipelines*, we held a workshop during which we identified some likely data sources needed for our data analytics project, but now, we need to dive deeper to gather additional information.

We need to identify who owns each data source that we plan to ingest, and then do a deep dive with the data source owner and ask questions such as the following:

- What is the format of the data (relational database, NoSQL database, semi-structured data such as JSON or XML, unstructured data, and so on)?
- How much historical data is available?
- What is the total size of the historical data?
- How much new data is generated on a daily/weekly/monthly basis?
- Does the data currently get streamed somewhere, and if so, can we tap into the stream of data?
- How frequently is the data updated (constantly, such as with a database or streaming source, or on a scheduled basis, such as at the end of the day or when receiving a daily update from a partner)?
- How will this data source be used to add value to the business, either now or in the future?

Learning more about the data will help you determine the best service to use to ingest the data, and help with the initial sizing of services and estimating a budget.

Now that we have reviewed how to take an inventory of the data sources you want to use, let's move on to examining specific ingestion types, starting with ingestion from relational databases.

Ingesting data from a relational database

A common source of data for analytical projects is data that comes from a relational database system such as MySQL, PostgreSQL, SQL Server, or an Oracle database. Organizations often have multiple siloed databases, and they want to bring the data from these varied databases into a central location for analytics.

It is common for these projects to include ingesting historical data that already exists in the database, as well as syncing ongoing new and changed data from the database. There are a variety of tools that can be used to ingest from database sources, as we will discuss in this section.

AWS DMS

The primary AWS service for ingesting data from a database is **AWS DMS**, though there are other ways to ingest data from a database source. As a data engineer, you need to evaluate both the source and the target to determine which ingestion tool will be best suited.

AWS DMS is intended for doing either one-off ingestion of historical data from a database or replicating change data (often referred to as **Change Data Capture** (**CDC**)) from a relational database on an ongoing basis. When using AWS DMS, the target is either a different database engine (such as an Oracle-to-PostgreSQL migration), or an Amazon S3-based data lake. In this section, we will focus on ingesting data from a relational database to an Amazon S3-based data lake.

We introduced the AWS DMS service in *Chapter 3*, *The AWS Data Engineer's Toolkit*, so make sure you have read the *Overview of Amazon Database Migration Service (DMS)* section in that chapter to get a good understanding of how the service works.

While AWS DMS was originally a managed service only (meaning that DMS provisioned one or more EC2 servers as replication instances), AWS announced an AWS DMS serverless option in June 2023. When using the AWS DMS serverless option, DMS automatically sets up, scales, and manages the underlying compute resources, helping to simplify the database migration process.

AWS Glue

AWS Glue, a Spark processing engine that we introduced in *Chapter 3*, *The AWS Data Engineer's Toolkit*, can make connections to several data sources. This includes connections to JDBC sources, and custom connectors available on the AWS Glue Marketplace, enabling Glue to connect to many different database engines, and through those connections transfer data for further processing.

AWS Glue is well suited to certain use cases related to ingesting data from databases. Let's take a look at some of them.

Full one-off loads from one or more tables

AWS Glue can be configured to make a connection to a database and download data from tables. Glue effectively does a `select *` from the table, reading the table contents into the memory of the Spark cluster. At that point, you can use Spark to write out the data to Amazon S3, optionally in an optimized format such as Apache Parquet.

Initial full loads from a table, and subsequent loads of new records

AWS Glue has a concept called *job bookmarks*, which enables Glue to keep track of which data was previously processed, and then on subsequent runs only process new data. Glue does this by having you identify a column (or multiple columns) in the table that will serve as a bookmark key. The values in this bookmark key must always increase in value, although gaps are allowed.

For example, if you have an audit table that has a `transaction_ID` column that sequentially increases for each new transaction, then this would be a good fit for ingesting data with AWS Glue while using the bookmark functionality.

The first time the job runs, it will load all records from the table and store the highest value for the `transaction_ID` column in the bookmark. For illustration purposes, let's assume the highest value on the initial load was `944,872`. The next time the job runs, it effectively does a `select * from audit_table where transaction_id > 944872`.

Note that this process is unable to detect updated or deleted rows in the table, so it is not well suited to all use cases. An audit table, or similar types of tables where data is always added to

the table but existing rows are never updated or deleted, is the optimal use case for this functionality.

Creating AWS Glue jobs with AWS Lake Formation

AWS Lake Formation includes several blueprints to assist in automating some common ingestion tasks. One of these ingestion blueprints allows you to use AWS Glue to ingest data from a database source. With a few clicks in the Lake Formation console, you can configure your ingest requirements (one-off versus scheduled, full table load or incremental load with bookmarks, and so on). Once configured, Lake Formation creates the Glue job for ingesting from the database source, the Glue Crawlers for adding newly ingested data into the Glue Data Catalog, and Glue workflows for orchestrating the different components.

Other ways to ingest data from a database

There are several other approaches to ingesting data from a database to an Amazon S3-based data lake, which we will cover briefly in this section.

Amazon EMR provides a simple way to deploy several common Hadoop framework tools, and some of these tools are useful for ingesting data from a database. For example, you can run Apache Spark on Amazon EMR and use a JDBC driver to connect to a relational database to load data into the data lake (in a similar way to our discussion about using AWS Glue to connect to a database).

If you are running MariaDB, MySQL, or PostgreSQL on Amazon **Relational Database Service (RDS)**, you can use RDS functionality to export a database snapshot to Amazon S3. This is a fully managed process that writes out all tables from the snapshot to Amazon S3 in Apache Parquet format. This is the simplest way to move data into Amazon S3 if you are using one of the supported database engines on RDS, and you want all tables exported to S3 on a scheduled basis. For more information, see the documentation at `https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_ExportSnapshot.html`.

There are also several third-party commercial tools, many containing advanced features that can be used to move data from a relational database to Amazon S3 (although these often come at a premium price). This includes tools such as Qlik Replicate (previously known as Attunity), a well-known tool for moving data between a wide variety of sources and targets (including relational databases, data warehouses, streaming sources, enterprise applications, cloud providers, and legacy platforms such as DB2).

You may also find that your database engine contains tools for directly exporting data in a flat format that can then be transferred to Amazon S3. Some database engines also have more advanced tools, such as Oracle GoldenGate, a solution that can generate CDC data as a Kafka stream.

Note, however, that these tools are often licensed separately and can add significant additional expense. For an example of using Oracle GoldenGate to generate CDC data that has been loaded into an S3 data lake, search for the AWS blog post titled *Extract Oracle OLTP data in real time with GoldenGate and query from Amazon Athena,* or find it here: `https://aws.amazon.com/blogs/big-data/extract-oracle-oltp-data-in-real-time-with-goldengate-and-query-from-amazon-athena/`.

A reminder about CDC

We introduced the concept of CDC in *Chapter 3*, *The AWS Data Engineer's Toolkit*, but it is an important concept, so here is a reminder. When rows in a relational database are deleted or updated, there is no practical way to capture those changes using standard database query tools (such as SQL). But when replicating data from a database to a new source, it is important to be able to identify those changes so that they can be applied to the target. This process of identifying and capturing these changes (new inserts, updates, and deletes) from the database log files is referred to as CDC.

Deciding on the best approach to ingesting from a database

While all these tools can be used in one way or another to ingest data from a database, there are several points to consider when deciding on the best approach for your specific use case.

The size of the database

If the total size of the database tables you want to load is large (many tens of GB or larger), then doing a full nightly load would not be a good approach. The full load could take a significant amount of time to run and puts a heavy load on the source system while running. In this scenario, a better approach is to do an initial load from the database and then constantly sync updates from the source using CDC, such as by using *AWS DMS*.

For very large databases, you can use AWS DMS with an Amazon Snowball device to load data to the Snowball device in your data center. Once the data has been loaded, you return the device to AWS, and it will load it to Amazon S3. AWS DMS will capture all CDC changes while the Snowball device is being transferred back to AWS so that once the data is loaded, you can create an ETL job to apply changes to the full data load. This initial load via Snowball may take a week or longer due to the physical transport of the device back to AWS, plus the application of a week or more of CDC data. However, once the initial load and application of CDC data are complete, CDC changes can continue to be applied in near real time.

For smaller databases, you can consider using RDS' snapshot export functionality, AWS Glue, or native database tools to load the entire database to Amazon S3 on a scheduled basis. This will often be the simplest and most cost-effective method, but it is not right for every use case, being best suited to smaller databases (low tens of GB or smaller) and where having a daily update, rather than near-real-time updates, meets requirements.

Database load

If you have a database with a consistent production load at all times, you will want to minimize the additional load you place on the server to sync to the data lake. In this scenario, you can use DMS to do an initial full load, ideally from a read replica of your database if it's supported as a source by DMS. For ongoing replication, DMS can use database log files to identify CDC changes, and this places a lower load on database resources.

Whenever you do a full load from a database (whether you're using AWS DMS, AWS Glue, or another solution), there will be a heavier load on the database as a full read of all tables is re-

quired. You need to consider this load and, where possible, use a read replica of your database for the full load. If using Amazon RDS, see the following documentation for details on how to configure a read replica: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_ReadRepl.html.

If a smaller database is running on Amazon RDS, the best solution would be to use the export-to-S3-from-snapshot functionality of RDS if it's supported for your database engine. This solution places no load on your source database; however, as it loads from a snapshot of the database, it cannot be used for real-time data replication, but rather only for scheduled exports.

Data ingestion frequency

Some analytic use cases are well suited to analyzing data that is ingested on a fixed schedule (such as every night). However, some use cases will want to have access to new data as fast as possible.

If your use case requires access to data coming from a database source as soon as possible, then using a service such as AWS DMS to ingest CDC data is the best approach. However, remember that CDC data just indicates what data has changed (new rows having been inserted and existing rows updated or deleted), so you still require a process to apply those changes to the existing data to enable querying for the most up-to-date state.

If your use case allows for regularly scheduled updates, such as nightly, you can do a scheduled full load (if the database's size and performance impacts allow), or you can have a nightly process to apply the CDC data that was collected during the day to the previous snapshot of data.

In *Chapter 7, Transforming Data to Optimize for Analytics*, we will review several approaches for applying CDC data to an existing dataset.

Technical requirements and compatibility

When evaluating different approaches to and tools for ingesting data from a database source, it is very important to involve the database owner and admin team upfront to technically evaluate the proposed solution.

A data engineering team may decide on a specific toolset upfront, based on its requirements and its broad understanding of compatibility with the source systems. However, at the time of implementation, it may discover that the source database team objects to certain security or technical requirements of the solution, and this can lead to significant project delays.

For example, AWS DMS supports CDC for several MySQL versions. However, DMS does require that binary logging is enabled on the source system with specific configuration settings for CDC to work.

Another example is that AWS DMS does not support server-level audits when SQL Server 2008/2008 R2 is used as a source. Certain commands related to enabling this functionality will cause DMS to fail.

It is critical to get the buy-in of the database owner and admin team before finalizing a solution. All of these requirements and limitations are covered in the AWS DMS documentation (and other solutions or products should have similar documentation covering their requirements). Reviewing these requirements, in detail, with the database admin team upfront is critical to the success of the project.

In the next section, we will take a similar look at tools for and approaches to ingesting data from streaming sources.

Ingesting streaming data

An increasingly common source of data for analytic projects is data that is continually generated and needs to be ingested in near real time. Some common sources of this type of data are as follows:

- Data from IoT devices (such as smartwatches, smart appliances, and so on)
- Telemetry data from various types of vehicles (cars, airplanes, and so on)
- Sensor data (from manufacturing machines, weather stations, and so on)
- Live gameplay data from mobile games
- Mentions of the company brand on various social media platforms

For example, Boeing, the aircraft manufacturer, has a system called **Airplane Health Management** (**AHM**) that collects in-flight airplane data and relays it in real time to Boeing systems. Boeing processes the information and makes it immediately available to airline maintenance staff via a web portal.

In this section, we will look at several tools and services for ingesting streaming data, as well as things to consider when planning for streaming ingestion.

Amazon Kinesis versus Amazon Managed Streaming for Kafka (MSK)

The two primary services for ingesting streaming data within AWS are *Amazon Kinesis* and *Amazon MSK*. Both of these services were described in *Chapter 3*, *The AWS Engineer's Toolkit*, so ensure you have read those sections before proceeding.

In summary, both Amazon Kinesis and Amazon MSK are services from AWS that offer the ability to write streaming data to storage, and then have data consumers connect to that storage to read the messages. This is commonly used as a way to decouple applications producing streaming data from applications that are consuming data. Both services can scale up to handle millions of messages per second.

In this section, we will examine some of the primary differences between the two services and look at some of the factors that contribute to deciding which service is right for your use case.

Serverless services versus managed services

Amazon Kinesis Data Streams is available in two capacity modes – either *provisioned* or *on-demand*. With provisioned mode, you need to configure the number of shards (the base through-

put unit of a Kinesis data stream) that is provisioned for the stream, while in on-demand mode, the number of shards is automatically allocated and managed by Kinesis.

In both modes, AWS manages the underlying compute resources for you. Therefore, with Kinesis, you never have to select EC2 instance sizes or EBS storage or deal with any of the underlying resources. You either select to provision and manage the number of shards for the Kinesis stream on your own (with provisioned mode) or you select to have AWS manage that for you with on-demand mode.

With *Amazon Kinesis Data Firehose*, much like Amazon Kinesis Data Streams in on-demand mode, the compute automatically scales up and down in response to message throughput changes without requiring any configuration.

In a similar way, *Amazon MSK* is also available in both a provisioned and a serverless mode. With provisioned mode, AWS manages the infrastructure for you, but you still need to be aware of and make decisions about the underlying compute infrastructure and software. For example, you need to select from a list of EC2 instance types to power your MSK cluster, configure VPC network settings, configure storage size and options, select the version of Apache Kafka to deploy, and fine-tune a range of Kafka configuration settings.

With Amazon MSK Serverless, the deployment is simpler as you do not need to make decisions about the EC2 instance size, storage, and version, but you also have less control over the Kafka cluster options.

If you have a team with existing skills in using Apache Kafka, and you need to fine-tune the performance of the stream, then you may want to consider MSK provisioned mode. If you're just getting started with streaming and your use case does not have a requirement to fine-tune performance, then Amazon Kinesis or Amazon MSK serverless mode may be a better option.

Open-source flexibility versus proprietary software with strong AWS integration

Amazon MSK is a managed version of Apache Kafka, a popular open-source solution. Amazon Kinesis is proprietary software created by AWS, although there are some limited open-source elements, such as Kinesis Agent.

With Apache Kafka, there is a large community of contributors to the software, and a large ecosystem providing a diverse range of connectors and integrations. Kafka provides out-of-the-box integration with hundreds of event sources and event sinks (including AWS services such as Amazon S3, but also many other popular products, such as PostgreSQL, Elasticsearch, and others).

With Amazon Kinesis, AWS provides strong integration with several AWS services, such as Amazon S3, Amazon Redshift, and Amazon OpenSearch Service. Kinesis also provides integration with a number of external services such as Splunk, DataDog, MongoDB, Sumo Logic, New Relic, and others through Amazon Kinesis Data Firehose.

When deciding between the two services, ensure that you consider the types of integrations your use case requires and how that matches with the out-of-the-box functionality of either

Kinesis or MSK.

At-least-once messaging versus exactly once messaging

When working with streaming technologies, some use cases have specific requirements around how many times messages may be processed by data consumers. Amazon Kinesis and Apache Kafka (and therefore Amazon MSK) provide different guarantees around message processing.

Amazon Kinesis provides an *at-least-once* message processing guarantee. This effectively guarantees that every message generated by a producer will be delivered to a consumer for processing. However, in certain scenarios, a message may be delivered more than once to a consuming application, introducing the possibility of data duplication.

With **Apache Kafka** (and therefore Amazon MSK), as of version 0.11, the ability to configure your streams for *exactly-once* message processing was introduced. When you configure your Apache Kafka stream, you can configure the `processing.guarantee=exactly_once` setting to enable this.

With Amazon Kinesis, you need to build the logic for anticipating and appropriately handling how individual records are processed multiple times in your application. AWS provides guidance on this in the Kinesis documentation, in the *Handling Duplicate Records* section.

If your use case calls for a guarantee that all messages will be delivered to the processing application exactly once, then you should consider Amazon MSK. Amazon Kinesis is still an option, but you will need to ensure your application handles the possibility of receiving duplicate records.

A single processing engine versus niche tools

Apache Kafka is most closely compared to Amazon Kinesis Data Streams as both provide a powerful way to consume streaming messages. While both can be used to process a variety of data types, Amazon Kinesis does include several distinct sub-services for specialized use cases.

For example, if your use case involves ingesting streaming audio or video data, then Amazon Kinesis Video Streams is custom-designed to simplify this type of processing. Or, if you have a simple use case of wanting to write out ingested streaming data to targets such as Amazon S3, Amazon OpenSearch Service, or Amazon Redshift (as well as some third-party services), then Amazon Kinesis Data Firehose makes this task simple.

Deciding on a streaming ingestion tool

There are several factors to consider when deciding on which AWS service to use for processing your streaming data, as we covered in this section. Both Amazon Kinesis and Amazon MSK Serverless require minimal upfront configuration and ongoing maintenance, while Amazon MSK in provisioned mode provides the ability to fine-tune your cluster performance and options.

Amazon Kinesis has a subset of services for special use cases, so you should evaluate your use case against the various Kinesis services and see if one of these will meet your current and expected future requirements. If your use case has specific requirements, such as exactly once message delivery, the ability to fine-tune the performance of the stream, or needs integration with third-party products not directly available in Kinesis, then consider Amazon MSK. Finally, you should also compare the pricing for each service based on your requirements.

In the next few sections, you will get hands-on with ingesting data from a database using AWS DMS and then ingesting streaming data using Amazon Kinesis.

Hands-on – ingesting data with AWS DMS

As we discussed earlier in this chapter, AWS DMS can be used to replicate a database into an Amazon S3-based data lake (among other uses). Follow the steps in this section to do the following:

1. Deploy a CloudFormation template that configures a MySQL RDS instance and then deploys an EC2 instance to load a demo database into MySQL.
2. Set up a DMS replication instance and configure endpoints and tasks.
3. Run the DMS instance in full-load mode.
4. Run a Glue Crawler to add the tables that were newly loaded into S3 into the AWS Glue Data Catalog.
5. Query the data with Amazon Athena.
6. Delete the CloudFormation template in order to remove the resources that have been deployed.

NOTE

The following steps assume the use of your AWS account's default VPC and security group. You will need to modify the steps as needed if you're not using the default.

Deploying MySQL and an EC2 data loader via CloudFormation

AWS CloudFormation is a service that enables you to deploy infrastructure as code. Using CloudFormation templates provides the ability to deploy AWS services using either JSON- or YAML-formatted templates. This enables you to treat your infrastructure as code, meaning you can store the templates in a version control system (to manage changes to templates and integrate code reviews), and easily deploy infrastructure reliably and repeatably via CI/CD pipelines.

To get started, download the CloudFormation template from this book's GitHub site at `https://github.com/PacktPublishing/Data-Engineering-with-AWS-2nd-edition/blob/main/Chapter06/mysql-ec2loader.cfn`.

Save the file locally to your computer by right-clicking on **Raw** and then selecting **Save Link As** in your browser window.

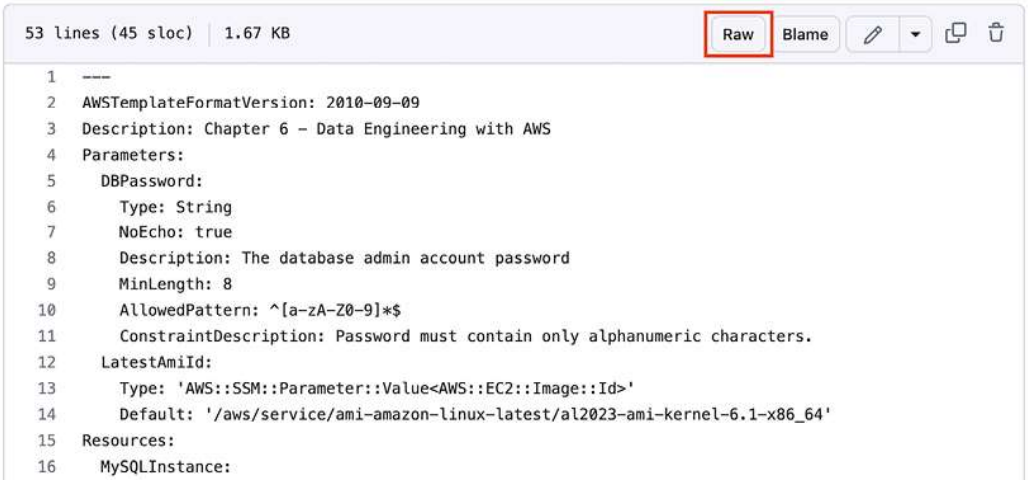

Figure 6.2: Download the CloudFormation template from GitHub

In the following steps, we are going to deploy a CloudFormation template that will do the following:

1. Request that the user provides a password (`DBPassword`) that will be set as the admin password for the MySQL instance.
2. Set a parameter (`LatestAmiId`) that provides a path to an *AWS Secrets Manager* entry that contains the **Amazon Machine Image (AMI)** ID for deploying an instance with Amazon Linux 2023 within which the region the user is working.
3. Create an Amazon RDS MySQL instance (`MySQLInstance`) with 20 GB of allocated storage, using the `db.t3.micro` instance type. Set the admin password for the instance to the value entered by the user as captured in the `DBPassword` parameter.
4. Create an Amazon EC2 instance (`EC2Instance`) of type `t3.micro` , provide user data that will run commands when the instance is launched to download a MySQL demo database, and write the database schema and data to the MySQL instance. Include the `DependsOn` option to indicate that the `MySQLInstance` resource must be created prior to this resource being created (this is to ensure that the MySQL instance is ready to receive the data that will be written via the EC2 instance we launch here).

Use the following steps to deploy the CloudFormation template:

1. Log in to the **AWS Management Console** (`https://console.aws.amazon.com`) and ensure that you are in the region that you have used for all the hands-on activities in this book.
2. In the top search bar, search for and select `CloudFormation` to access the CloudFormation console.
3. Click on **Create stack, With new resources (standard)**.
4. Under the **Specify template** section, select **Upload a template file**, click on **Choose file**, and select the `mysql-ec2loader.cfn` file you downloaded from GitHub.
5. Click on **Next**.
6. For **Stack name**, provide a name (such as `dataeng-aws-chapter6-mysql-ec2`).
7. In the **Parameters** section, provide a `DBPassword` that will be used as the password for your MySQL admin user.
8. Leave the **LatestAmiId** field with the default string, and then click **Next**.
9. Leave all other defaults and click **Next**.

10. Leave all defaults, and click on **Submit**.

CloudFormation will now take a few minutes to deploy your MySQL RDS instance and the EC2 instance that will download the demo database (called `SakilaDB`), and then load the demo database to the MySQL instance that was just created.

Once the deployment is finished, the stack status will change to **CREATE_COMPLETE**, as shown in the following screenshot.

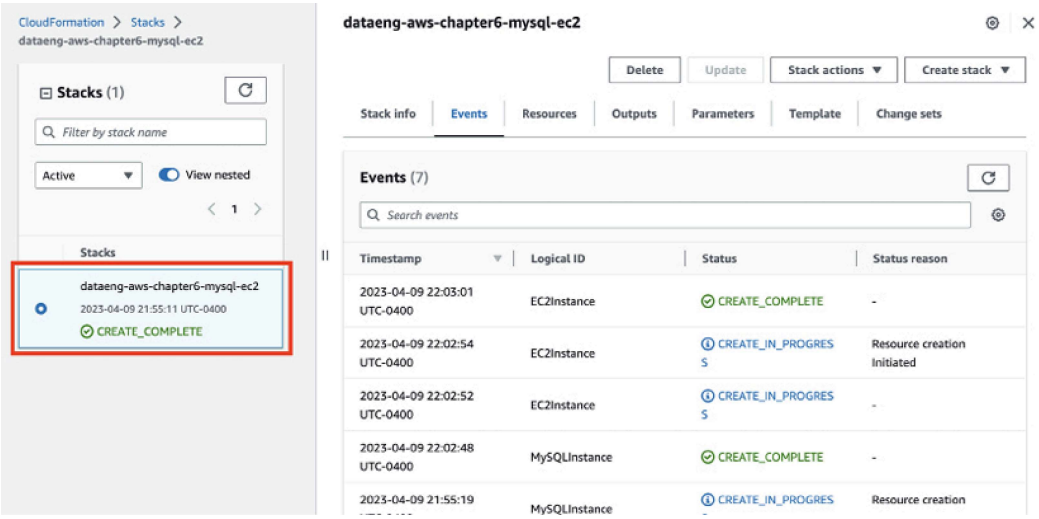

Figure 6.3: CloudFormation stack successfully created

You can continue with the next step to create an IAM policy and role for DMS to use while the deployment is still running. But you must ensure the deployment completes successfully before configuring DMS.

Creating an IAM policy and role for DMS

In this section, we will create an IAM policy and role that will allow DMS to write to our target S3 bucket:

1. In the AWS Management Console, search for and select **IAM** using the top search bar.
2. In the left-hand menu, click on **Policies** and then click **Create policy**.
3. By default, the **Visual editor** is selected, so change to text entry by clicking on the **JSON** tab.
4. Replace the boilerplate code in the text box with the following policy definition (which can also be copied from this book's GitHub site). Make sure you replace `<initials>` in the bucket name with the correct landing zone bucket name you created in *Chapter 2*, *Data Management Architectures for Analytics*:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:*"
```

```
            ],
            "Resource": [
                "arn:aws:s3:::dataeng-landing-zone-<initials>",
                "arn:aws:s3:::dataeng-landing-zone-<initials>/*"
            ]
        }
    ]
}
```

This policy grants permissions for all S3 operations (get, put, and so on) for the `dataeng-landing-zone-<initials>` bucket. This will give DMS the permissions needed to write out CSV files in the landing zone bucket with the data from our MySQL database.

5. Click **Next**.
6. Provide a descriptive policy name, such as `DataEngDMSLandingS3BucketPolicy` , and click **Create policy**:

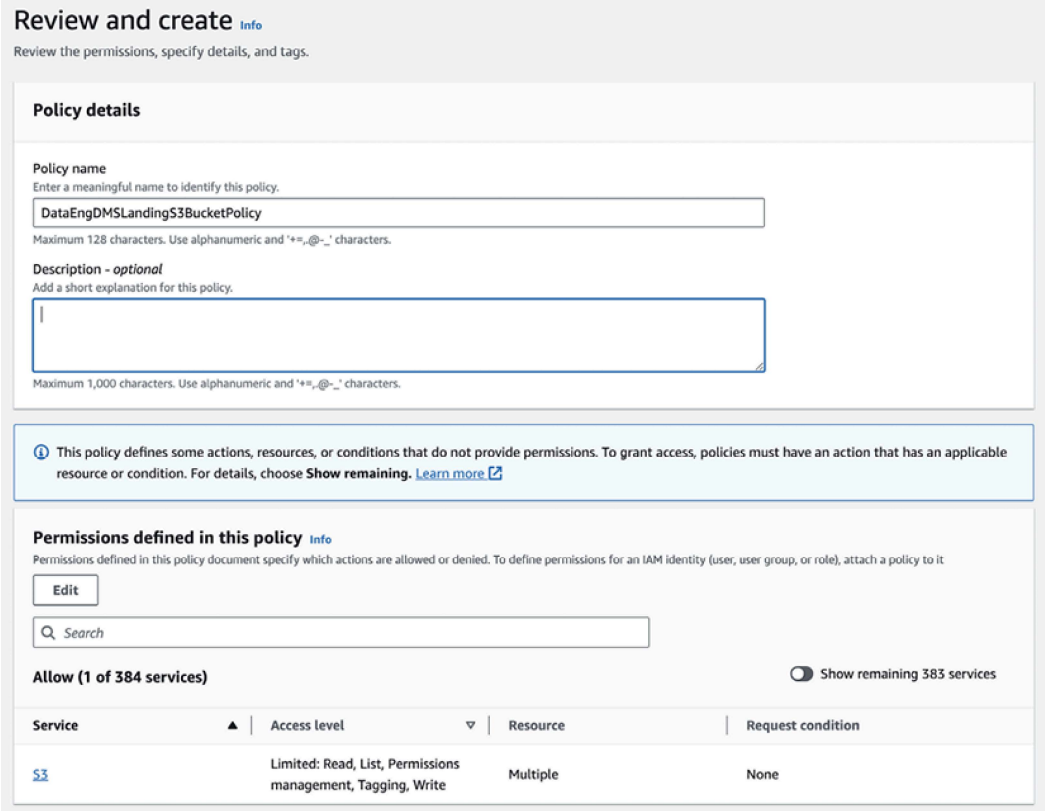

Figure 6.4: Creating an IAM policy to grant S3 permissions

7. In the left-hand menu, click on **Roles** and then click **Create role**.
8. For **Trusted entity type**, make sure **AWS service** is selected.
9. For **Use case**, search for and select **DMS**, make sure to click the selector for the DMS service, and then click **Next**.

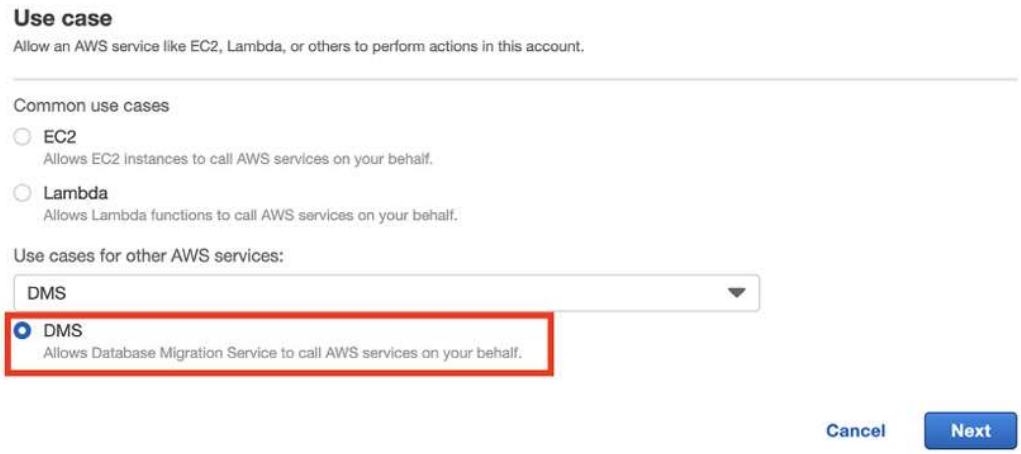

Figure 6.5: Creating a new role for the DMS service to use

10. Search for and select the policy you created in *step 6* (such as `DataEngDMSLandingS3BucketPolicy`), and then click **Next**.
11. Provide a descriptive **Role name**, such as `DataEngDMSLandingS3BucketRole`, and click **Create role**.
12. Click on the newly created role and copy and paste the role **Amazon Resource Name (ARN)** property somewhere where you can easily access it; it will be required in the next section.

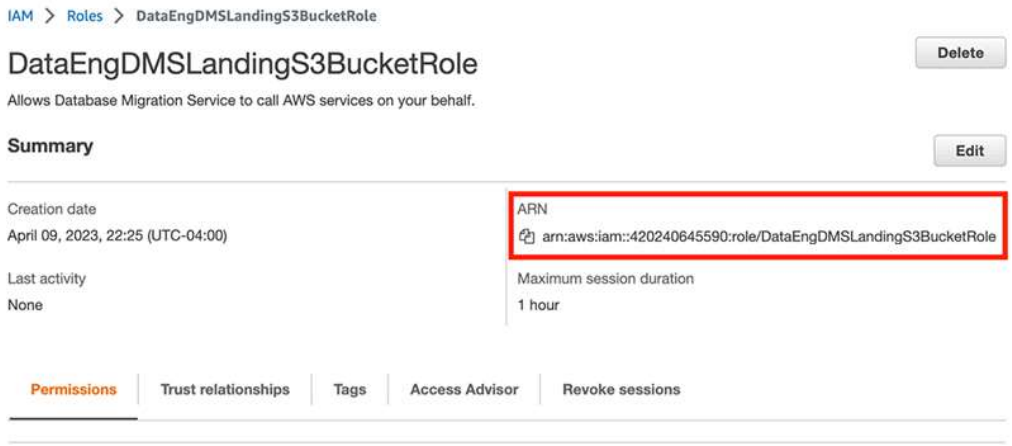

Figure 6.6: Capturing the ARN of the newly created role

Now that we have created the required IAM permissions, we will create a DMS replication instance, as well as other required DMS resources (such as source and target endpoints, as well as a database migration task).

Configuring DMS settings and performing a full load from MySQL to S3

In this section, we will create a DMS replication instance (a managed EC2 instance that connects to the source endpoint, retrieves data, and writes to the target endpoint), and also configure the source and target endpoints. We will then create a database migration task that pro-

vides the configuration settings for the migration. **Make sure that your CloudFormation template has been deployed completely before continuing with this section.**

In the following steps, you will configure DMS and start the full-load job:

1. In the AWS Management Console, search for **DMS** using the top search bar and click on **Database Migration Service**.
2. In the left-hand menu, click on **Replication Instances**.
3. At the top of the page, click on **Creation replication instance**.
4. Provide a **Name** for the replication instance; for example, `mysql-s3-replication`.
5. For **Instance class**, select `dms.t3.micro`.
6. For **High Availability**, select **Dev or test workload (Single-AZ)**.
7. For **Allocated storage**, enter `10` (the database we are replicating is very small, so 10 GB is enough space).
8. In the **VPC** dropdown, select the **default VPC**.
9. Leave everything else as the defaults and click **Create replication instance**. Note that it may take a few minutes for the replication instance to be created and ready.
10. In the left-hand menu, click on **Endpoints**.
11. At the top right, click on **Create endpoint**.
12. For **Endpoint type**, select **Source endpoint** and then click the box for **Select RDS DB Instance**.
13. For **RDS Instance**, use the drop-down list to select the MySQL database that was created by the CloudFormation template deployment.
14. Under **Endpoint configuration**, for **Access to endpoint database**, select **Provide access information manually**.
15. For **Password**, provide the password that you set for the database when deploying the CloudFormation template (*step 7* in the **Deploying the CloudFormation template** step)
16. Leave all other defaults and then click **Create endpoint** at the bottom right.
17. Now that we have created the source endpoint, we can create the target endpoint by clicking on **Create endpoint** at the top right.
18. For **Endpoint type**, select **Target endpoint**.
19. For **Endpoint identifier**, type in a name for the endpoint, such as `s3-landing-zone-sakila-csv`.
20. For **Target engine**, select **Amazon S3** from the drop-down list.
21. For **Amazon Resource Name (ARN) for service access role**, enter the ARN for the IAM role you recorded in *step 12* of the previous section.
22. For **Bucket name**, provide the name of the landing zone bucket you created in *Chapter 2*, *Data Management Architectures for Analytics* (for example, `dataeng-landing-zone-<initials>`).
23. For **Bucket folder**, enter `sakila-db`.
24. Expand the endpoint settings and click on **Add new setting**. Select `AddColumnName` from the settings list, and for the value, type `True`.

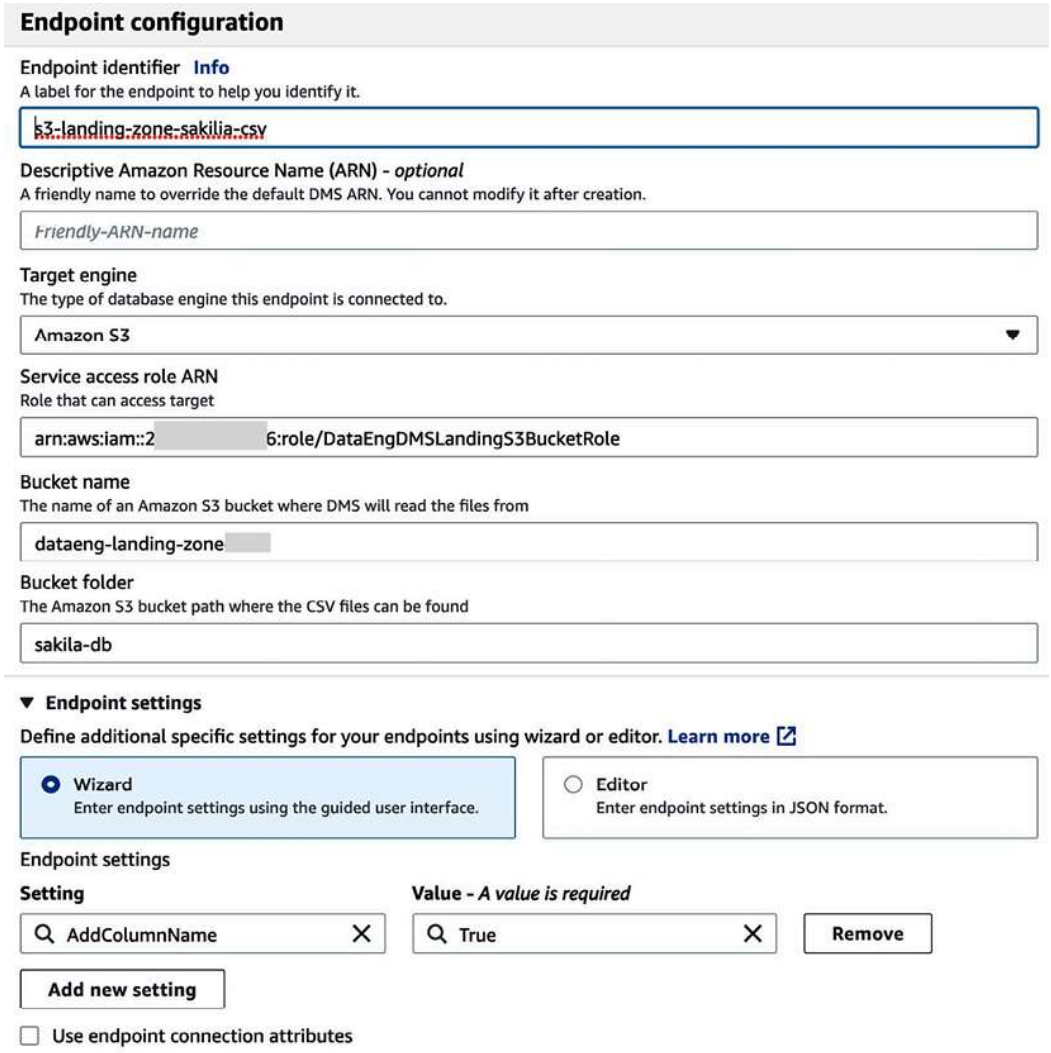

Figure 6.7: AWS DMS S3 target endpoint

25. Click **Create Endpoint**.
26. On the left-hand side, click **Database migration tasks**, and then click **Create task**.
27. For **Task identifier**, provide a descriptive name for the task, such as `dataeng-mysql-s3-sakila-task`.
28. For **Replication instance**, select the instance you created in *step 4* of the previous section, such as `mysql-s3-replication`.
29. For **Source database endpoint**, select the source endpoint that links to your MySQL instance.
30. For **Target database endpoint**, select the S3 target endpoint you created previously.
31. For **Migration type**, select **Migrate existing data** from the dropdown. This does a one-time migration from the source to the target.
32. Leave the defaults for **Task settings** as they are.
33. For **Table mappings**, under **Selection rules**, click on **Add new selection rule.**
34. For **Schema**, select **Enter a schema**. For **Source name**, enter `%sakila%`, and for **Source Table name**, leave it set as `%`.
35. Leave the defaults for **Selection rules** and all other sections as they are and click **Create task**.

36. Once the task has been created, the full load will be automatically initiated and the data will be loaded from your MySQL instance to Amazon S3. Click on the task identifier and review the **Table statistics** tab to monitor your progress.

Our previously configured S3 event for all CSV files written to the landing zone bucket will be triggered for each file that DMS loads. This will run the Lambda function we created in *Chapter 3*, which will create a new Parquet version of each file in the **CLEAN ZONE** bucket. This will also register each table in the AWS Glue Data Catalog.

Querying data with Amazon Athena

The Lambda function that was run for each CSV file created by DMS also registers each new Parquet file as part of a table in the AWS Glue database.

We can now query the newly ingested data using the Amazon Athena service.

1. First, we need to create a new Amazon S3 folder to store the results of our Athena queries. In the **AWS Management Console**, search for and select **S3** using the top search bar.
2. Click on **Create bucket**, and for **Bucket name**, enter `athena-query-results-<INITIALS>`. Replace `<INITIALS>` in the bucket name with a unique identifier, such as the one you have used with other buckets in previous chapters.
3. Make sure the **AWS Region** is set to the Region you have used for the previous hands-on exercises. Leave all other defaults and click on **Create bucket**.
4. In the **AWS Management Console**, search for and select **Athena** using the top search bar.
5. Within the Athena console, click on the **Settings** tab.
6. Click on **Manage** in the **Settings** tab, and for **Location of query result**, provide the path of the bucket we just created (such as `s3://athena-query-results-gse23`), and then click **Save**.
7. Return to the **Editor** tab, and then in the **Database** dropdown on the left-hand side, select `sakila` from the drop-down list.
8. In the **New query** window, run the following query: `select * from film limit 20;`.
9. This query returns the results of the first 20 fictional films in the Sakila database.
10. If your query runs successfully and returns 20 results, that confirms that your DMS task was completed successfully. Since the infrastructure we deployed does have a low cost per hour while it is running, we can save costs by deleting the DMS replication instance and the resources deployed by the CloudFormation template. Open up the **DMS** service console, and on the left-hand side, click on **Database migration tasks**. We need to delete the task before we can delete the associated replication instance, so select the task, and from the **Actions** menu, click **Delete**, and then confirm the deletion in the pop-up box.
11. Once the replication task has been deleted, on the left-hand side, click on **Replication instances**. Select the replication instance you created earlier, and then from the **Actions** menu, select **Delete**. Confirm that you want to delete the replication instance by clicking on **Delete** in the pop-up box.
12. Open up the **CloudFormation service console**, and click on **Stacks** in the left-hand menu.
13. **Select the stack** that you deployed earlier, and then click on **Delete**. Confirm the deletion by clicking on **Delete stack** in the popup.

Congratulations! You have successfully replicated a MySQL database into your S3-based data lake. To learn more about ingesting data from MySQL to Amazon S3, see the following AWS

documentation:

- Using Amazon S3 as a target for AWS Database Migration Service (`https://docs.aws.amazon.com/dms/latest/userguide/CHAP_Target.S3.html`)
- Using a MySQL-compatible database as a source for AWS DMS (`https://docs.aws.amazon.com/dms/latest/userguide/CHAP_Source.MySQL.html`)

Now that we have got hands-on with ingesting batch data from a database into our Amazon S3 data lake, let's look at one of the ways to ingest streaming data into our data lake.

Hands-on – ingesting streaming data

Earlier in this chapter, we looked at two options for ingesting streaming data into AWS, namely Amazon Kinesis and Amazon MSK. AWS provides an open-source solution for streaming sample data to Amazon Kinesis; therefore, in this section, we will use the Amazon Kinesis service to ingest streaming data. To generate streaming data, we will use the AWS open-source *Amazon Kinesis Data Generator* (*KDG*).

In this section, we will perform the following tasks:

1. Configure **Amazon Kinesis Data Firehose** to ingest streaming data, and write the data out to Amazon S3.
2. Configure **Amazon KDG** to create mock streaming data.

To get started, let's configure a new Kinesis Data Firehose instance to ingest streaming data and write it out to our Amazon S3 data lake.

Configuring Kinesis Data Firehose for streaming delivery to Amazon S3

Kinesis Data Firehose is designed to enable you to easily ingest data from streaming sources, and then write that data out to a supported target (such as Amazon S3, which we will do in this exercise). Let's get started:

1. In the AWS Management Console, search for and select **Kinesis** using the top search bar.
2. The Kinesis landing page provides links to create new streams using the Kinesis features of Kinesis Data Streams, Kinesis Data Firehose, or Kinesis Data Analytics. Select the Kinesis Data Firehose service, and then click on **Create delivery stream**.
3. In this exercise, we are going to use KDG to send data directly to Firehose, so for **Source**, select **Direct PUT** from the drop-down list. For **Destination**, select **Amazon S3** from the drop-down list.
4. For **Delivery stream name**, enter a descriptive name, such as `dataeng-firehose-streaming-s3`.
5. For the optional section of **Transform and convert records**, leave both options **unchecked. Transform source records with AWS Lambda** functionality can be used to run data validation tasks or perform light processing on incoming data with AWS Lambda, but we want to ingest the data without any processing, so we will leave this disabled. **Convert record format** can be used to convert incoming data into Apache Parquet or Apache ORC format. However, to do this, we would need to specify the schema of the in-

coming data upfront. We are going to ingest our data without changing the file format, so we will leave this disabled.

6. For **S3 bucket**, select the landing zone bucket you created previously; for example, `s3://dataeng-landing-zone-<initials>`.
7. For **Dynamic Partitioning**, leave this option set to **Not enabled**.
8. By default, Kinesis Data Firehose writes the data to S3 with a prefix to split incoming data by `YYYY/MM/dd/HH`. For our dataset, we want to load streaming data into a **streaming** prefix, and we only want to split data by the year and month that it was ingested. Therefore, we must set **S3 bucket prefix** to `streaming/!{timestamp:yyyy/MM/}`. For more information on custom prefixes, see **https://docs.aws.amazon.com/firehose/latest/dev/s3-prefixes.html**.
9. If we set a custom prefix for incoming data, we must also set a custom error prefix. Set **S3 bucket error output prefix** to `!{firehose:error-output-type}/!{timestamp:yyyy/MM/}`.
10. Expand the **Buffer hints, compression and encryption** section.
11. The S3 buffer conditions allow us to control the parameters for how long Kinesis buffers incoming data before writing it out to our target. We specify both a buffer size (in MB) and a buffer interval (in seconds), and whichever is reached first will trigger Kinesis to write to the target. If we used the maximum buffer size of 128 MB and a maximum buffer interval of 900 seconds (15 minutes), we would see the following behavior. If we receive 1 MB of data per second, Kinesis Data Firehose will trigger after approximately 128 seconds (when 128 MB of data has been buffered). On the other hand, if we receive 0.1 MB of data per second, Kinesis Data Firehose will trigger after the 900-second maximum buffer interval. For our use case, we will set **Buffer size** to `1 MB` and **Buffer interval** to `60` seconds.
12. For all the other settings, leave the default settings as they are and click on **Create delivery stream**.

Our Kinesis Data Firehose stream is now ready to receive data. So, in the next section, we will generate some data to send to the stream using the **KDG** tool.

Configuring Amazon Kinesis Data Generator (KDG)

Amazon **KDG** is an open-source tool from AWS that can be used to generate customized data streams and can send that data to Kinesis Data Streams or Kinesis Data Firehose.

The Sakila database we previously loaded was for a company that produced classic movies and rented those out of its DVD stores. The DVD rental stores went out of business years ago, but the owners have now made their classic movies available for purchase and rental through various streaming platforms.

The company receives information about its classic movies being streamed from its distribution partners in real time, in a standard format. Using KDG, we will simulate the streaming data that's received from partners, including the following:

- The streaming timestamp
- Whether the customer rented, purchased, or watched the trailer
- `film_id` that matches the Sakila film database
- The distribution partner's name
- The streaming platform
- The state that the movie was streamed in

KDG is a collection of HTML and JavaScript files that run directly in your browser and can be accessed as a static site in GitHub. To use KDG, you need to create an Amazon Cognito user in your AWS account, and then use that user to log in to KDG on the GitHub account.

AWS has created an Amazon **CloudFormation** template that you can deploy in your AWS account to create the required Amazon Cognito user. This CloudFormation template creates an AWS Lambda function in your account to perform the required setup.

Follow these steps to deploy the CloudFormation template, create the required Cognito user, and configure KDG:

1. Open the KDG help page in your browser by going to https://awslabs.github.io/amazon-kinesis-data-generator/web/help.html.
2. Read the information about how the CloudFormation template works to create Cognito credentials in your account. When you're ready, click on the **Create a Cognito User with CloudFormation** button.
3. The AWS Management Console will open to the CloudFormation **Create Stack** page. **IMPORTANT**: When opening the link, the Region may default to Oregon (us-west-2-), therefore, change **Region** in the console to the Region you are using for the exercises in this book, then accept the CloudFormation defaults, and click **Next**.

4. On the **Specify stack details** page, provide a **Username** and **Password** for your Cognito user and click **Next**.
5. For **Configure stack options**, leave all the default settings as they are and click **Next**.
6. Review the details of the stack to be created, and then click the box to acknowledge that **AWS CloudFormation may create IAM resources**. Then, click **Submit**.
   Refresh the web page and monitor it until the stack's status is `CREATE_COMPLETE`.

7. Once the stack has been successfully deployed, go to the **Outputs** tab and take note of the `KinesisDataGeneratorUrl` value. Click on the link and open a new tab.
8. Use the **username** and **password** you set as parameters for the CloudFormation template to log in to the Amazon KDG portal.
9. Set **Region** to the same Region in which you created the Kinesis Data Firehose delivery stream. If you need a mapping of Region IDs to region names, refer to the following documentation:
   https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.RegionsAndAvailabilityZones.html.
10. For **Stream/delivery stream**, from the dropdown, select the Kinesis Data Firehose stream you created in the previous section.
11. For **Records per second**, set this as a **constant** of `10` records per second. Leave **Compress records unchecked**.
12. For the record template, we want to generate records that simulate what we receive from our distribution partners. Paste the following into the Template 1 section of KDG (this can also be copied and pasted from the GitHub site for this chapter at https://github.com/PacktPublishing/Data-Engineering-with-AWS-2nd-edition/tree/main/Chapter06):

```
{
    "timestamp":"{{date.now}}",
```

```
    "eventType":"{{random.weightedArrayElement(
      {
        "weights": [0.3,0.1,0.6],
        "data": ["rent","buy","trailer"]
      }
        )}}",
    "film_id":{{random.number(
        {
            "min":1,
            "max":1000
        }
    )}},
   "distributor":"{{random.arrayElement(
        ["amazon prime", "google play", "apple itunes", "vudo", "fandango now", "microsoft", "youtu
    )}}",
    "platform":"{{random.arrayElement(
        ["ios", "android", "xbox", "playstation", "smart tv", "other"]
    )}}",
    "state":"{{address.state}}"
}
```

13. Click **Send data** to start sending streaming data to your **Kinesis Data Firehose** delivery stream. Because of the configuration that we specified for our Firehose stream, the data we are sending is going to be buffered for 60 seconds, and then a batch of data is written to our landing zone S3 bucket. This will continue for as long as we leave KDG running.
14. Allow KDG to send data for at least 5-10 minutes (3,000-6,000 records), and then click on **Stop Sending Data to Kinesis**. The longer this process runs, the more data you will have when querying this dataset in later chapters. If you want a larger dataset, consider leaving this process to run for 30-60 minutes before stopping sending data to Kinesis.

> By leaving KDG running for at least 5-10 minutes, it will have created enough data for us to use in later chapters, where we will join this data with data we migrated from our MySQL database. This is enough data to run the subsequent exercises, but you can leave KDG running for longer if you want a larger dataset to work with. However, this will cause your ETL jobs to run for longer, queries will take longer, etc.

We can now use a **Glue crawler** to create a table in our AWS Glue Data Catalog for the newly ingested streaming data.

Adding newly ingested data to the Glue Data Catalog

In this section, we will run a **Glue crawler** to examine the newly ingested data, infer the schema, and automatically add the data to the Glue Data Catalog. Once we do this, we can query the newly ingested data using services such as **Amazon Athena**. Let's get started:

1. In the AWS Management Console, search for and select **Glue** using the top search bar.
2. In the left-hand menu, click on **Crawlers** (under **Data Catalog**).
3. Click on **Create crawler**.
4. Enter a descriptive name for **Name**, such as `dataeng-streaming-crawler` , and click **Next**.
5. For **Data source configuration**, under **Data sources**, click on **Add a data source.**

6. Make sure **Data source** is set to **S3**, and for **Location of S3 data**, set the S3 path to your `dataeng-landing-zone-<initials>` bucket, and add a suffix of `streaming`. For example, `s3://dataeng-landing-zone-<initials>/streaming/`, but make sure to replace `<initials>` with the unique identifier you used, and make sure to include the ending slash after the suffix.
7. Leave all other settings as default, and then click on **Add an S3 data source**.
8. Click on **Next**.
9. For **Configure security settings**, click on **Create new IAM role**. Provide a suffix for the IAM role name, such as `AWSGlueServiceRole-streaming-crawler`, and then click **Create**.
10. Click **Next**.
11. For **Output configuration**, click on **Add database**. In the new tab that opens, provide a descriptive database name, such as `streaming_db`, and then click **Create database**.
12. Go back to your browser tab with the **Glue console**, and click the **refresh icon** to the right of **Target database**. From the dropdown, you should then be able to select the newly created `streaming_db` database. Leave the other settings as default (such as keeping the schedule as **On demand**), and then click **Next**.
13. Review the settings on the **Review and create** page, and then click **Create crawler**.
14. Select your new crawler from the list and click **Run**.

When the crawler finishes running, it should have created a new table for the newly ingested streaming data.

Querying the data with Amazon Athena

Now that we have ingested our new streaming data, and added the data to the AWS Glue Data Catalog using the AWS Glue crawler, we can query the data using Amazon Athena:

1. In the AWS Management Console, search for and select **Athena** using the top search bar.
2. On the left-hand side, from the **Database** drop-down list, select the database you created in the previous step (such as `streaming_db`).
3. In the query window, type in `select * from streaming limit 20`.

The result of the query should show 20 records from the newly ingested streaming data, matching the pattern that we specified for KDG. Note how the Glue Crawler automatically added the two fields that we configured as our partitions (year and month).

Summary

In this chapter, we reviewed several ways to ingest common data types into AWS. We reviewed how AWS DMS and AWS Glue can be used to ingest data from a relational database to S3, and how Amazon Kinesis and Amazon MSK can be used to ingest streaming data.

In the hands-on section of this chapter, we used both the AWS DMS and Amazon Kinesis services to ingest data and then used AWS Glue to add the newly ingested data to the AWS Glue Data Catalog and query the data with Amazon Athena.

In the next chapter, *Chapter 7*, *Transforming Data to Optimize for Analytics*, we will review how we can transform the ingested data to optimize it for analytics, a core task for data engineers.

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