10
Orchestrating the Data Pipeline

Throughout this book, we have discussed various services that can be used by data engineers to ingest and transform data, as well as make it available for consumers. We looked at how we could ingest data via **Amazon Kinesis Data Firehose** and **AWS Database Migration Service (DMS)**, and how we could run **AWS Lambda** and **AWS Glue** functions to transform our data. We also discussed the importance of updating a data catalog as new datasets are added to a data lake, and how we can load subsets of data into a data mart or data warehouse for specific use cases.

For the hands-on exercises, we made use of various services, but for the most part, we triggered these services manually. However, in a real production environment, it would not be acceptable to have to manually trigger these tasks, so we need a way to automate various data engineering tasks. This is where data pipeline orchestration tools come in.

Modern-day ETL applications are designed with a modular architecture to facilitate the use of the best purpose-built tool to complete a specific task. A data engineering pipeline (also sometimes referred to as a workflow) stitches all of these components together to create an ordered execution of related tasks, which can then be triggered to automatically run on a given schedule, or in response to another event occurring.

To build our pipeline, we need an orchestration engine to define and manage the sequence of tasks, as well as the dependencies between tasks. The orchestration engine also needs to be intelligent enough to perform different actions based on the failure or success of a task, and should be able to define and execute tasks that run in parallel, as well as tasks that run sequentially.

In this chapter, we will look at how to manage data pipelines with different orchestration engines. First, we will examine some of the core concepts of pipeline orchestration, and then review several different options within AWS for orchestrating data pipelines.

In the hands-on activity for this chapter, we will orchestrate a data pipeline using the AWS Step Functions service.

In this chapter, we will cover the following topics:

- Understanding the core concepts for pipeline orchestration
- Examining the options for orchestrating pipelines in AWS
- Hands-on – orchestrating a data pipeline using AWS Step Functions

Technical requirements

To complete the hands-on exercises in this chapter, you will need an AWS account where you have access to a user with administrator privileges (as covered in *Chapter 1, An Introduction to Data Engineering*). We will make use of various AWS services, including **AWS Lambda**, **AWS Step Functions**, and **Amazon Simple Notification Service** (**SNS**).

You can find the code files of this chapter in the GitHub repository using the following link: `https://github.com/PacktPublishing/Data-Engineering-with-AWS-2nd-edition/tree/main/Chapter10`

Understanding the core concepts for pipeline orchestration

In *Chapter 5*, *Architecting Data Engineering Pipelines*, we architected a high-level overview of a data pipeline. We examined potential data sources, discussed the types of data transformations that may be required, and looked at how we could make transformed data available to our data consumers.

Then, we examined the topics of data ingestion, transformation, and how to load transformed data into data marts in more detail in the subsequent chapters. As we discussed previously, these steps are often referred to as an **Extract, Transform, Load** (**ETL**) process.

We have now come to the part where we need to combine the individual steps involved in our ETL processes to operationalize and automate how we process data. But before we look deeper at the AWS services to enable this, let's examine some of the key concepts around pipeline orchestration.

What is a data pipeline, and how do you orchestrate it?

A simple definition is that a **data pipeline** is a collection of data processing tasks that need to be run in a specific order. Some tasks may need to run sequentially, while other tasks may be able to run in parallel. You could also refer to the sequencing of these tasks as a **workflow**.

Data pipeline orchestration refers to automating the execution of tasks involved in a data pipeline workflow, managing dependencies between the different tasks, and en-

suring that the pipeline runs when it is meant to.

Think of the data pipeline as the smallest entity for performing a specific task against a dataset. For example, if you receive data from a partner regularly, your first data pipeline may involve validating that the data that's received is valid, and then converting the data file into an optimized format, such as **Parquet**. If you have hundreds of partners sending you data files, then this same pipeline may run for each of those partners.

You may also have a second data pipeline that runs at a specific time of day that validates that the data from all your partners has been received, and then runs a Spark job to join the datasets and enrich the data with additional proprietary data.

Once that data pipeline finishes running, you may have a third pipeline that loads the newly enriched data into a data warehouse.

While you could place all of these steps in a single data pipeline, it is a recommended best practice to split pipelines into the smallest logical grouping of steps. In the preceding example of processing files we receive from our partners throughout the day, our first step is getting newly received files converted into Parquet format, but we only want to do that if we can confirm that the file we received is valid. As such, we group those two tasks (confirming that the file is valid, and then converting into Parquet format) into our first pipeline. The goal of our second pipeline is to join the files we received from our partners throughout the day and enrich the new file with additional data. However, our second pipeline should also include a step to validate and report on whether all the expected partner files were received.

Let's explore some of the common concepts that are regularly used when developing data pipelines.

What is a directed acyclic graph?

When talking about data pipelines, you may hear the term **directed acyclic graph**, commonly referred to as **DAG**. If you Google this term, you may find a lot of complex mathematical explanations of what a DAG is. This is because this term not only applies to data pipelines, but is used to define many different types of ordered processes. For example, DAGs are also used to design compilers.

A simple explanation of a DAG is that it represents connections between nodes, with the flow between nodes always occurring in only one direction and never looping back to an earlier node (acyclic means not a cycle).

The following diagram shows a simple DAG:

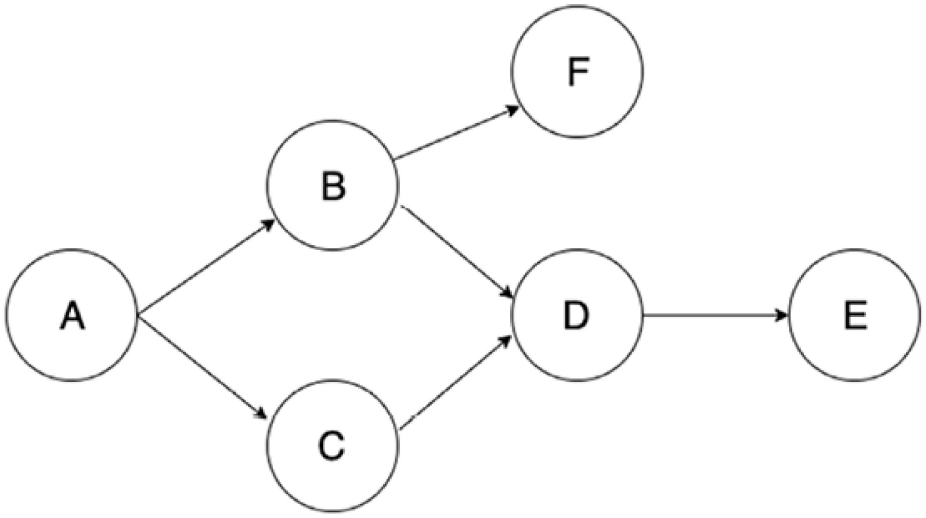

Figure 10.1: A simple example of a directed acyclic graph

If this DAG represented a data pipeline, then the following would take place:

- When event A completes, it triggers event B and event C.
- When event B completes, it triggers event F.
- When events B and C are complete, they trigger event D.
- When event D completes, it triggers event E.

In the preceding example, event F could never loop back to event A, B, or C, as that would break the acyclic part of the DAG definition.

No rule says that data pipelines have to be defined as DAGs, although certain orchestration tools do require this. For example, **Apache Airflow** (which we will discuss in more detail later in this chapter) requires pipelines to be defined as a DAG, and if there is a cycle in a pipeline definition where a node loops back to a previous node, this would not run.

However, **AWS Step Functions** does allow for loop cycles in the definition of a state machine, so Step Functions-based pipelines do not enforce that the pipeline should be a DAG.

How do you trigger a data pipeline to run?

There are two primary types of triggers for a pipeline – **schedule-based pipelines** and **event-based pipelines**.

Traditionally, pipelines were all triggered on a schedule. This could be once a day, every hour, or perhaps even every 15 minutes. This is still a common approach, especially for batch-orientated pipelines. In our prior pipeline example, the second pipe-

line could be an example of a scheduled pipeline that runs once per day to join and enrich partner files that are received throughout the day.

Today, however, a lot of pipelines are created to be **event-driven**. In other words, the pipeline is triggered in response to some specific event being completed. Event-based workflows are useful for reducing the latency between data becoming available and the pipeline processing that data. For example, if you expect that you will have received the data files you need at some point between 4 A.M. and 6 A.M., you could schedule the pipeline to run at 6 A.M. However, if all the data is available by 5 A.M. on some days, using an event-based trigger can get your pipeline running earlier.

In our earlier example of a pipeline, the first pipeline (to validate files and convert to Parquet format) would be an event-driven pipeline that runs in response to a partner having uploaded a new file. Within AWS, there is strong support for creating event-driven activities, such as triggering an event (which could be a pipeline) based on a file being written to a specific Amazon S3 bucket (and we will get hands-on with an event-driven pipeline in the hands-on section of this chapter).

Using manifest files as pipeline triggers

A **manifest** is often used to refer to a list of cargo carried by a ship, or other transport vehicles. The manifest document may be reviewed by agents at a border crossing or port to validate what is being transported.

In the world of data pipelines, a common concept is to create a **manifest file** that contains information about other files that form part of a batch of files.

In our data pipeline example of receiving files from our partners, we may find that the partner sends hundreds of small CSV files in a batch every hour. We may decide that we do not want to run our pipeline on each file that we receive, but instead to process all the small CSV files in a batch together and convert them into a single Parquet file.

In this case, we could instruct our partners to send a manifest file at the end of each batch of files that they send to us. This manifest file would list the name of each file that's transferred, as well as potentially some validation data, such as file size, or a calculated SHA-256 hash of the file.

We could then configure our S3 event notification to only trigger when a file that begins with the name `manifest` is written to our bucket. When this happens, we will trigger our pipeline to run, and perhaps the first step in our pipeline would be to read the manifest file, and then for each file listed in the manifest, verify that it exists. We could also calculate the SHA-256 hash of the file, and verify that it matches what is

listed in the manifest. Once the files have been verified, we could run our ETL job to read in all the files and write the files out in Parquet format.

This process would still be considered an event-driven pipeline, even though we are not responding to every file upload event, just the completion of a batch of uploads, as represented in the manifest file.

There will, of course, be times when a job will fail, and we need to make sure that we build error handling into our pipelines, as discussed next.

How do you handle the failures of a step in your pipeline?

As part of the orchestration process to automate the processing of steps in a pipeline, we need to ensure that failures are handled correctly. Therefore, it is also important that log files related to each step of the pipeline are easily accessible. In this section, we will look at some important concepts involved in failure handling and logging.

Common reasons for failure in data pipelines

There are many reasons why a specific step in a data pipeline may fail. Some common reasons for errors include the following:

- **Data quality issues**: If one of the steps in your pipeline expects to receive CSV files to process, but instead receives a file in JSON format that it does not know how to process, this would lead to a hard failure (that is, a failure that your job cannot recover from until the data quality issue is resolved).
- **Code errors**: When you update a job, it is possible to introduce a syntax, or logic, error into the code. Testing your code before deploying it into production is very important, but there may be times when your testing does not catch a specific error. This would also be a hard failure, requiring you to redeploy fixed code.
- **Endpoint errors**: One of the steps in your pipeline may involve the need to either read or write data to or from a specific endpoint (such as reading a file in S3 or writing data into a data warehouse). At times, your processing job may not be able to connect to the endpoint, and there may be a number of reasons for this type of error. For example, there could be a temporary network error preventing the connection from being successful, or it could be because of insufficient permissions to access a target database. If it were a temporary network error, this could be considered a soft failure (that is, one that may be overcome by retrying the step). But if it is due to insufficient permissions, the error would be considered a hard failure, and there is no point immediately retrying the step, as you will need the permissions issue to be resolved first.
- **Dependency errors**: Data pipelines generally consist of multiple steps with complex dependencies. This includes dependencies within the pipeline, as well as de-

pendencies between different pipelines. If your job is dependent on a previous step, then the job it is dependent on is referred to as an upstream job. If your job fails, any jobs that depend on it are considered downstream jobs. Dependency errors can be hard failures (such as an upstream job or pipeline having a hard failure) or soft failures (e.g., the upstream job is taking longer than expected to complete, but if you retry your step, it may complete later).

Hard failures generally interrupt processing (and are also likely to cause failures in downstream jobs) until someone takes a specific action to resolve the error. When a hard failure occurs, a data engineer, or operations person, will need to examine the log files to identify the error message, and then take corrective action (such as getting access configured if the error indicated a permissions failure).

Soft failures (such as intermittent networking issues), however, can benefit from having a good retry strategy, as we will discuss next.

Pipeline failure retry strategies

When you're designing your pipeline, you should consider implementing a retry strategy for failed steps. Many orchestration tools (such as Apache Airflow and AWS Step Functions) will allow you to specify the number of retries, the interval between retry attempts, as well as a backoff rate.

The **retry backoff rate** (also known as **exponential backoff**) causes the time between retry attempts to be increased on each retry. With AWS Step Functions, for example, you can specify a `BackOffRate` value that will multiply the delay between retries by that value. For example, if you specify a retry interval of 10 seconds and a backoff rate of 1.5, Step Functions will wait 15 seconds (10 seconds x 1.5) for the second retry, 22.5 seconds (15 seconds x 1.5) for the third retry, and so on.

Having reviewed some of the core concepts of data pipelines and orchestration, we can now examine the tools that are available in AWS for creating and orchestrating pipelines.

Examining the options for orchestrating pipelines in AWS

As you will have noticed throughout this book, AWS offers many different building blocks for architecting solutions. When it comes to pipeline orchestration, AWS provides native serverless orchestration engines with **AWS Data Pipeline** and **AWS Step Functions**, a managed open-source project with **Amazon Managed Workflows for Apache Airflow** (**MWAA**), and service-specific orchestration with **AWS Glue workflows**.

There are pros and cons to using each of these solutions, depending on your use case. When making a decision on this, there are multiple factors to consider, such as the level of management effort, the ease of integration with your target ETL engine, logging, error-handling mechanisms, cost, and platform independence.

In this section, we'll examine each of the four pipeline orchestration options.

AWS Data Pipeline (now in maintenance mode)

AWS Data Pipeline is one of the oldest services that AWS has for creating and orchestrating data pipelines, having been originally released in 2012. However, this service is now in maintenance mode and it is not recommended to build new ETL pipelines using this service.

In December 2022, AWS updated the Data Pipeline documentation to encourage customers to migrate to alternative data integration services, such as AWS Glue, AWS Step Functions, or Amazon MWAA. If you have existing pipelines that use this service, refer to the AWS documentation for a guide on how to migrate to a more modern service at `https://docs.aws.amazon.com/datapipeline/latest/DeveloperGuide/migration.html`.

Let's now take a look at the first of those services that are recommended as an alternative to AWS Data Pipeline, the AWS Glue service.

AWS Glue workflows to orchestrate Glue resources

In *Chapter 3, The AWS Data Engineer's Toolkit*, we introduced **AWS Glue workflows** as a feature of the AWS Glue service. As a reminder, AWS Glue enables you to easily build and run Spark-and Python-based ETL jobs, and the Glue workflows feature can be used to build a pipeline to orchestrate the running of Glue components (Glue Crawlers and Glue ETL jobs).

For use cases where you create a data pipeline that only uses AWS Glue components, the use of Glue workflows can be a good fit. For example, you could create the following pipeline using Glue workflows:

- Run a Glue Crawler to add CSV files that have been ingested into a new partition to the Glue Data Catalog.
- Run a Glue Spark job to read the new data using the catalog, and then transform the CSV files into Parquet files.
- Run another Glue Crawler to add the newly transformed Parquet files to the Glue Data Catalog.

- Run two Glue jobs in parallel. One Glue job aggregates data and writes the results into a DynamoDB table. The other Glue job creates a newly enriched dataset that joins the new data to an existing reference set of data.
- Run another Glue Crawler to add the newly enriched dataset to the Glue Data Catalog.
- Run a Glue Python Shell job to send a notification about the success or failure of the job.

While a fairly complex data pipeline can be created using Glue workflows (as demonstrated above), many use cases require the use of other AWS services, such as EMR for running Hive jobs, or writing files to an SQS queue. While Glue workflows do not support integration with non-Glue services directly, it is possible to run a Glue Python Shell job that uses the Boto3 library to interact with other AWS services. However, this is not as feature-rich or as obvious to monitor as interacting with those services directly.

Glue workflows are a good fit for those pipelines that only use AWS Glue, but other options should be considered if you want to orchestrate additional services outside of the Glue family of services. If your use case only uses AWS Glue services, then the following section will be helpful to understand some best practices for using AWS Glue workflows.

Monitoring and error handling

Glue workflows includes a graphical UI that can be used to monitor job progress. With the UI, you can see whether any step in the pipeline has failed, and you can also resume the workflow from a specific step once you have resolved the issue that caused the error. While the Glue workflows feature does not include a retry mechanism as part of the workflow definition, you can specify the number of retries in the properties of individual Glue jobs.

CloudWatch Events provides a real-time stream of change events that can be generated by some AWS services, including AWS Glue. While Glue does not generate any events related to Glue workflows directly, events are generated from individual Glue jobs. For example, there is a *Glue Job State Change* event that is generated for Glue jobs that reflects one of the following states: `SUCCEEDED`, `FAILED`, `TIMEOUT`, or `STOPPED`.

Using Amazon EventBridge, you can automate actions to take place when a new event and status you are interested in is generated. For example, you can create an EventBridge rule that picks up Glue job `FAILED` events, and then triggers a Lambda function to run, which sends an email notification with details of the failure.

Triggering Glue workflows

When you create a Glue workflow, you can select the mechanism that will cause the workflow to run. There are three ways that a Glue workflow run can be started.

If set to **on-demand**, the workflow will only run when it's started manually from the console, or when it's started using the Glue API or CLI.

If set to **scheduled**, you can specify a frequency for running the job, such as hourly, daily, monthly, or for specific days of the week (such as Mondays to Fridays). Alternatively, you can set a custom schedule using a `cron` expression, which uses a string to set a frequency to run. For example, if you set the `cron` expression to `*/30 8-16 * * 2-6`, the workflow will run *every 30 minutes between 8 A.M and 4:59 P.M., Mondays to Fridays*.

Glue workflows also support an **event-driven approach**, where the workflow is triggered in response to an EventBridge event. With this approach, you can configure an Amazon EventBridge rule to send events to a Glue workflows as the target, such as an S3 PutObject event for a specific S3 bucket and prefix.

When configuring your workflow, you can also specify triggering criteria, where you specify that you only want the workflow to run after a certain number of events are received, optionally specifying a maximum amount of time to wait for those events.

For example, if you have a business partner that sends many small `.csv` files throughout the day, you may not want to process each file individually, but rather process a batch of files. For this use case, you can configure the workflow to trigger once 100 events have been received and specify a time delay of 3,600 seconds (1 hour).

This time delay starts when the first unprocessed event is received. If the specified number of events is not received within the time delay you entered, the workflow will start anyway and process the events that have been received.

If you receive 100 events between 8 A.M. and 8:40 A.M., the first run of the workflow will be triggered at 8:40 A.M. If you receive only 75 events between 8:41 A.M. and 9:41 A.M., the workflow will run a second time at 9:41 A.M. anyway and process the 75 received events, since the time delay of 1 hour has been reached.

While Glue workflows can be an ideal service for pipelines that only use Glue services, if you are looking for a more comprehensive solution that can also orchestrate other AWS services and on-premises tools, then you should consider AWS Step Functions or Apache Airflow, which we will discuss next.

Apache Airflow as an open-source orchestration solution

Apache Airflow is open-sourced orchestration software, originally developed at Airbnb, that provides functionality for authoring, monitoring, and scheduling workflows. Some of the features available in Airflow include stateful scheduling, a rich user interface, core functionality for logging, monitoring, and alerting, and a code-based approach to authoring pipelines.

Within AWS, a managed version of Airflow is available as a service called **Amazon Managed workflows for Apache Airflow** (**MWAA**). This service simplifies the process of getting started with Airflow, as well as the ongoing maintenance of Airflow infrastructure, since the underlying infrastructure is managed by AWS. Like other AWS managed services, AWS ensures the scalability, availability, and security of the Airflow software and infrastructure. Please refer to the overview of Amazon MWAA in *Chapter 3, The Data Engineer's Toolkit*, for more information on the architecture of this managed service.

When deploying the managed MWAA service in AWS, you can choose from multiple supported versions of Apache Airflow. At the time of writing, Airflow v1.10.12 and Airflow v2.6.3 are supported in the managed service.

Core concepts for creating Apache Airflow pipelines

Apache Airflow uses a code-based (Python) approach to authoring pipelines. This means that to work with Airflow, you do need some Python programming skills. However, having pipelines as code is a natural fit for saving pipeline resources in a source control system, and it also helps with creating automated tests for pipelines.

The following are some of the core concepts that are used to create Airflow pipelines.

Directed acyclic graphs (DAGs)

We introduced the concept of a **directed acyclic graph** (**DAG**) earlier in this chapter. In the context of Airflow, a data pipeline is created as a DAG (using Python to define the DAG), and the DAG defines the tasks in the pipeline, and the dependencies between the tasks.

In the Airflow user interface, you can also view a graphical representation of the DAG – the pipeline tasks and their dependencies, with tasks represented as nodes and arrows showing the dependencies between tasks.

Airflow Tasks

Airflow Tasks define the basic unit of work that a DAG performs. Each task is defined in a DAG with upstream and downstream dependencies, which defines the order in which the tasks should run. When a DAG runs, the tasks in the DAG move through various states, from `None` to `Scheduled`, to `Queued`, to `Running`, and then to `Success` or `Failed`.

Airflow Hooks

Airflow Hooks define how to connect to remote source and target systems, such as a database, or a system such as Zendesk. These Hooks contain the code that controls the connection to the remote system, and while Airflow includes several built-in hooks, it also lets you define custom hooks. With Amazon MWAA, default hooks are provided for many different AWS services (such as Amazon S3, AWS Glue, AWS Lambda, and more). There are also hooks available for various databases (such as Oracle, MySQL, and Postgres) and systems such as Slack.

Hooks contain the code to connect to remote systems, keeping that code separate from pipeline definitions.

Airflow Operators

Airflow Operators are predefined task templates that provide a pre-built interface for performing a specific task. Airflow includes several built-in core operators (such as `BashOperator` and `PythonOperator`, which execute a bash command or Python function).

There is also an extensive collection of additional operators that are released separately from Airflow Core. For example, the `LambdaInvokeFunctionOperator`, provided by AWS, can be used to invoke an AWS Lambda function.

Airflow Sensors

Airflow Sensors provides a special type of Airflow operator that is designed to wait until a specific action takes place. These Sensors regularly check whether the activity they are waiting on has been completed, and can be configured to time out after a certain period.

Using Airflow Sensors enables you to create event-driven pipelines. For example, you could use S3KeySensor, which waits for a specific key to be present on an S3 path and, once present, triggers a specific DAG to run.

Airflow Connections

Airflow Connections define the configuration information needed to connect to a remote system. For example, a connection may define the URL/hostname, port, username, and password that is used to make a connection to a database. Airflow Connections can be used by hooks, operators and sensors to define authentication credentials for connecting to external systems (such as a database, Amazon S3, or an AWS Lambda function).

Apache Airflow is a popular choice for creating and managing complex data pipelines, and has strong built-in support for AWS and third-party services. However, there is a fixed infrastructure cost for the service, as this is a managed service. Let's now look at AWS Step Functions, a serverless pipeline orchestration solution.

AWS Step Functions for a serverless orchestration solution

AWS Step Functions is a comprehensive serverless orchestration service that uses a low-code approach to develop data pipelines and serverless applications. Step Functions provides a powerful visual design tool that allows you to create pipelines with a simple drag and drop approach. Or, if you prefer, you can define your pipeline using **Amazon States Language** (**ASL**) directly using JSON.

AWS has built optimized, easy-to-use integrations between many different AWS services and Step Functions. For example, in the Step Functions interface you can easily add a step that runs a Lambda function, and select the name of the Lambda function to run from a drop-down list.

Step Functions also makes it easy to specify how to handle the failure of a state with custom retry policies, lets you specify catch blocks to catch specific errors, and takes custom actions based on the error. However, Step Functions does not currently support the ability to restart a state machine from a specific step.

For services where AWS has not built an optimized integration, you can still run the service by using the AWS SDK integration built into Step Functions. For example, there is no direct Step Functions integration for running Glue Crawlers, but you can add a state that calls the `Glue StartCrawler` API and specify the parameters that are needed by that API call.

In this next section, we review an example of a Step Functions state machine.

A sample Step Functions state machine

With Step Functions, you create a **state machine** that defines the various tasks that make up your data pipeline. Each task is considered a state within the state machine, and you can also have states that control the flow of your pipeline, such as a **choice**

state that executes a branch of the pipeline, or a **wait state** to pause the pipeline for a certain period.

When you're executing a Step Functions state machine, you can pass in a payload that can be accessed by each state. Each state can also add additional data to the payload, such as a status code indicating whether a task succeeded or failed.

The following diagram shows a sample state machine in Step Functions:

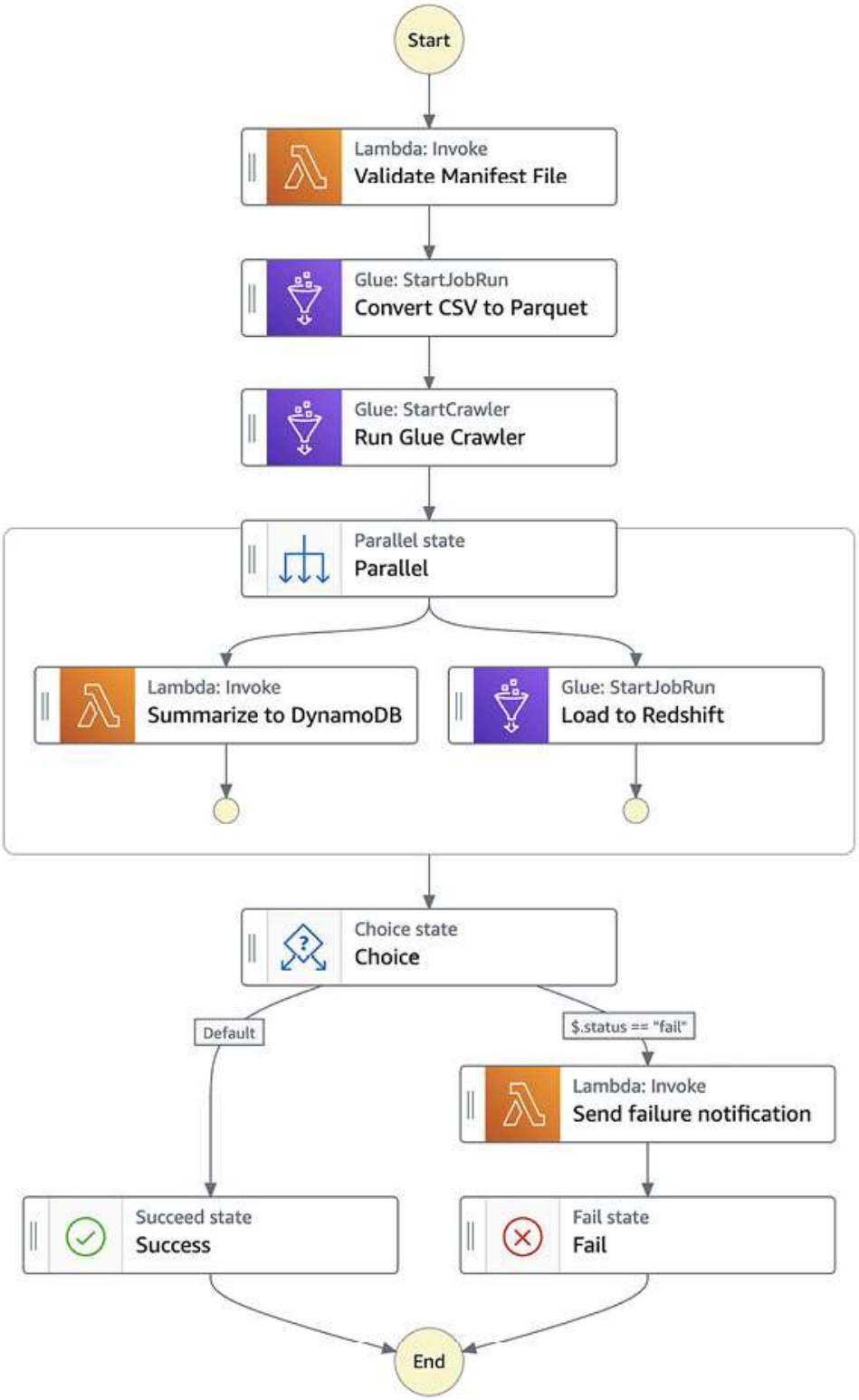

Figure 10.2: Sample Step Functions state machine

In this state machine definition, we can see the following states:

1. We start with a **Task state** that executes a Lambda function that validates a manifest file that has been received (ensuring that all the files listed in the manifest exist, for example).

2. We then have another **Task state**, this time to execute a Glue Job that will convert the files we received from CSV format into Parquet format.
3. Our next step is another **Task state**. This time, the task executes a Glue Crawler to update our data catalog with the new Parquet dataset we have generated.
4. We then enter a **Parallel state**, which is used to create parallel branches of execution in our state machine. In this case, we execute a Lambda function to summarize data from the Parquet file, and store the results in a DynamoDB table. At the same time, we trigger a Glue job to load the new data into our Redshift data warehouse.
5. We then enter a **Choice state**. The choice state specifies rules that get evaluated to determine what to do next. In this case, if our Lambda and Glue jobs succeeded, we end the state machine with a **Success state**. If either of them failed, we run a Lambda function to send a failure notification, and we end the state machine with a **Fail state**.

The visual editor that can be used in the console to create a state machine ultimately ends up generating an **Amazon States Language** (**ASL**) JSON file that contains the definition of the pipeline. You can store the JSON definition file for your data pipeline in a source control system, and then use the JSON file in a CI/CD pipeline to deploy your Step Functions state machine. You can edit your pipeline using the GUI interface in the console, or by directly editing the JSON file. Any edits you make directly to the JSON file can be imported into the console, and visualized. This enables you to use a combination of both the visual editor and direct edits of the JSON file in order to manage your pipeline.

In the hands-on exercises for this chapter, you will get the opportunity to build out a data pipeline using AWS Step Functions. However, before we do that, let's summarize your choices for data pipeline orchestration within AWS.

Deciding on which data pipeline orchestration tool to use

As we have discussed in this chapter, there are multiple options for creating and orchestrating data pipelines within AWS. And while we have looked at the four different options offered by AWS directly, there are many other options from AWS partners that could also be considered.

As covered previously, AWS Data Pipeline is now in maintenance mode and so it cannot be used for any new projects. If your project only uses AWS Glue jobs and the Glue Crawler, then AWS Glue workflows may be a good option. However, for larger and more complex pipelines, it is worth examining both Amazon MWAA and AWS Step Functions.

The following tables show a comparison of Step Functions and Amazon MWAA based on several different key attributes:

Criteria	AWS Step Functions	Amazon Managed Workflows for Apache Airflow (MWAA)
Short description	Serverless AWS native orchestration service	Managed AWS service for open source Apache Airflow
Graphical pipeline development	Yes	No
Graphical run visualization	Yes	Yes
Error and retry single step	Yes	Yes
Re-run from failed step	Custom workaround	Yes
Open source community support	No	Yes
Cost	Usage-based cost that depends on the complexity of the workflow	Constant base infrastructure cost, plus worker costs that can scale up and down
Scalability	Highly scalable, fully automatic	Highly scalable, managed by user or autoscaling groups, and can be configured
Infrastructure management	No infrastructure management or provisioning as everything handled by AWS	Requires making choices about infrastructure, but AWS manages the infrastructure and software
Language for pipeline development	JSON (or use of visual designer)	Python
Serverless/ managed	Serverless	Managed
Integration	Seamlessly integrates with AWS services and manual integration with non-AWS services	Strong integration support for many AWS services, as well as extensive third-party services

Figure 10.3: Comparison of AWS Step Functions and Amazon MWAA

We do not have space to cover getting hands-on with both Amazon MWAA and AWS Step Functions, but since Step Functions is serverless and provides an easy-to-use visual designer, we will look at how to build an AWS Step Functions state machine in the next section.

Hands-on – orchestrating a data pipeline using AWS Step Functions

In this section, we will get hands-on with the AWS Step Functions service, which can be used to orchestrate data pipelines. The pipeline we're going to orchestrate is relatively simple, but Step Functions can also be used to orchestrate far more complex

pipelines with many steps. To keep things simple, we will only use Lambda functions to process our data, but you could replace Lambda functions with Glue jobs in production pipelines that need to process large amounts of data.

For our Step Functions state machine, let's start by running a Lambda function that checks the extension of an incoming file to determine the type of file. Once determined, we'll pass that information on to the next state, which is a `CHOICE` state. If it is a file type we support, we'll call a Lambda function to process the file, but if it's not, we'll send out a notification, indicating that we cannot process the file.

If the Lambda function fails, we'll send a notification to report on the failure; otherwise, we will end the state machine with a `SUCCESS` status. Once we've created our state machine, we will configure EventBridge to automatically trigger the state machine when a file is uploaded to a specific Amazon S3 path.

Let's get building!

Creating new Lambda functions

Before we can create our state machine, we need to create the Lambda functions that we will orchestrate. We will create three separate Lambda functions in this section.

Using a Lambda function to determine the file extension

Our first Lambda function will check the extension of the file that's uploaded to our Amazon S3 bucket. Once we have the extension, we will return that in a JSON payload. Let's get started:

1. Log in to **AWS Management Console** and navigate to the **AWS Lambda** service at `https://console.aws.amazon.com/lambda/home`. Make sure that you are in the Region that you used for all the exercises in this book.
2. Click on **Create function**.
3. Select **Author from scratch**. Then, for **Function name**, enter `dataeng-check-file-ext`.
4. For **Runtime**, select **Python 3.10**. Leave the defaults for **Architecture** and **Permissions** as-is and click **Create function**.
5. In the **Code source** block, replace any existing code with the following code. This code receives an EventBridge event when a new S3 file is uploaded and uses the metadata included within the event to determine the extension of the file:

```
import urllib.parse
import json
import os
print('Loading function')
def lambda_handler(event, context):
    print("Received event: " + json.dumps(event, indent=2))
    # Get the object from the event and show its content type
    bucket = event['detail']['bucket']['name']
    key = urllib.parse.unquote_plus(event['detail']['object']['key'], encoding='utf-8'
    filename, file_extension = os.path.splitext(key)
    print(f'File extension is: {file_extension}')
    payload = {
        "file_extension": file_extension,
        "bucket": bucket,
        "key": key
        }
    return payload
```

6. Click the **Deploy** button above the code block section to save and deploy your Lambda function.

Now, we can create a second Lambda function that will process the file we received. However, for this exercise, the code in this Lambda function will randomly generate failures.

Using Lambda to randomly generate failures

For this Lambda function, we will use a random number generator to determine whether to cause an error in the Lambda function or allow it to succeed. We will do this by generating a random number that will be either 0, 1, or 2 and then dividing our random number by 10. When the random number is 0, we will get a “divide by zero” error from our function. We do this so that we can explore how Step Functions is able to handle failures in a function.

Let’s get started:

1. Repeat *steps 1* to *5* of the previous section to create the first Lambda function, but this time, for **Function name**, enter `dataeng-random-failure-generator`.
2. In the **Code source** block, replace any existing code with the following code:

```
from random import randint
def lambda_handler(event, context):
    print('Processing')
    #Our ETL code to process the file would go here.
    #However for this exercise we will instead randomly
    #cause the function to succeed or fail
```

```
    value = randint(0, 2)
    # We now divide 10 by our random number.
    # If the random number is 0, our function will fail
    newval = 10 / value
    print(f'New Value is: {newval}')
    return(newval)
```

3. Click the **Deploy** button above the code block section.

We now have two Lambda functions that we can orchestrate in our Step Functions state machine. But before we create the state machine, we have a few additional resources to create.

Creating an SNS topic and subscribing to an email address

If there is a failure in our state machine, we want to be able to send an email notification about the failure. We can use the SNS service to send an email. To do this, we need to create an SNS topic that we will send the notification to. Then, we can subscribe one or more email addresses to that topic. Let's get started:

1. Navigate to the **Amazon SNS** service at `https://console.aws.amazon.com/sns`. Ensure that you are in the Region that you have used for all the exercises in this book.
2. In the menu on the left-hand side, click on **Topics**, and then **Create topic**.
3. For **Type**, select **Standard**.
4. For **Name**, enter `dataeng-failure-notification` .
5. Leave all the other items as-is and click on **Create topic**.
6. In the **Subscriptions** section, click **Create subscription**.
7. For **Protocol**, select **Email**.
8. For **Endpoint**, enter your email address. Then, click on **Create subscription**.
9. Access your email and look for an email from *no-reply@sns.amazonaws.com* (this may take a minute or two to arrive). Click the **Confirm subscription** link in that email. You need to do this to receive future email notifications from Amazon SNS.

We now have an SNS topic with a confirmed email subscription that can receive SNS notifications. Next, we will create our new Step Functions state machine.

Creating a new Step Functions state machine

Now, we can orchestrate the various components that we have created so far (our two Lambda functions and the SNS topic we will use for sending emails) using AWS Step Functions:

1. Navigate to the **AWS Step Functions** service at `https://console.aws.amazon.com/states/home`. Ensure that you are in the Region that you have used for all the exercises in this book.
2. In the left-hand menu, click on **State machines**, and then click on **Create state machine**.
3. Select **Blank** for the template.
4. This will show a visual editor with a **Start** block and an **End** block in the **Design** tab. On the left-hand side, we have the components that we can use to design our state machine. Drag the **AWS Lambda Invoke** block into the visual designer, between the **Start** and **End** blocks.
5. On the right-hand side of the screen, set **State name** to `Check File Extension`.
6. Under **API Parameters**, use the drop-down list to set the **Function name** to be the name of the Lambda function that extracts the file extension (such as `dataeng-check-file-ext:$LATEST`).
7. On the right-hand side, click on the **Output** tab, and make sure the selector for **Filter output with OutputPath** is selected, and that the value is set to `$.Payload`. This option configures our **Check File Extension** state to have an output of whatever was returned by our Lambda function (in our case, we have configured our Lambda function to return a JSON payload that contains the S3 bucket, object, and file extension of the file to process).

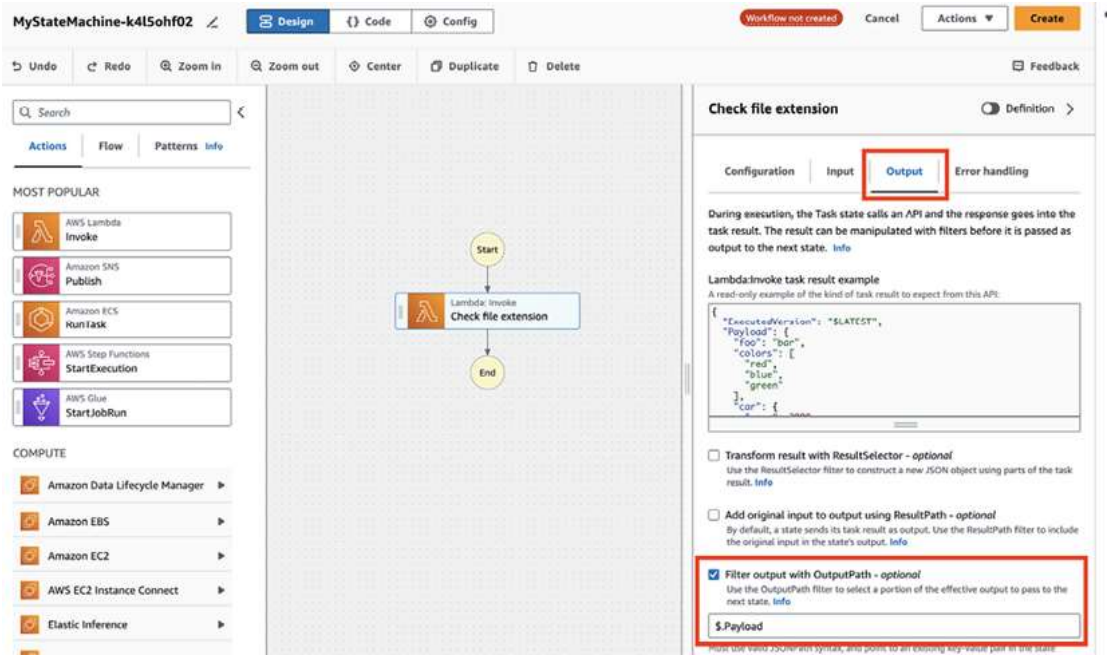

Figure 10.4: Building out a Step Functions state machine

8. On the left-hand side, click on the **Flow** tab. Then, drag the **Choice** state between the **Lambda Invoke** function and the **End** state. We use the **Choice** state to branch out our pipeline to run different processes, based on the output of a previous state.

In this case, our pipeline will do different things depending on the extension of the file we are processing.

9. On the right-hand side, under **Configuration** for our new choice state, click the **Pencil Edit** icon next to **Rule #1** and then click **Add conditions**.
10. On the pop-up screen, under **Variable**, enter `$.file_extension` (our Lambda function returns some JSON, including a JSON path of `file_extension` that contains a string with the extension of the file we are processing). Set **Operator** to **matches string** and for **value**, enter `.csv`. Then, click **Save conditions**.
11. On the left-hand side, switch back to the **Actions** tab and drag the **AWS Lambda Invoke** state to the **left-hand side of the two choice boxes**.
12. For our new **Lambda Invoke** state, set **State name** to **Process CSV** (since our **Choice** function is going to invoke this Lambda for any file that has an extension of `.csv`, as we set in *step 10*).
13. Under **API Parameters**, use the dropdown to set **Function name** to our second Lambda function (`dataeng-random-failure-generator:$LATEST`). In a real pipeline, we would have a Lambda function (or Glue job) that would read the CSV file that was provided as input and process the file. In a real pipeline, we may have also added additional rules to our **Cshoice** state for other file types (such as XLS or JPG) and had different Lambda functions or Glue jobs invoked to handle each file type. However, in this exercise, we are only focusing on how to orchestrate pipelines, so our Lambda function code is designed to simply divide 10 by a random number, resulting in random failures when the random number is 0.

14. On the left-hand side, switch back to the **Flow** tab and drag the **Pass** state to the **Default** rule box leading from our **Choice** state. The default rule is used if the output of our Lambda function does not match any of the other rules. In this case, our only other rule is for handling files with a `.csv` extension, so if a file has any other extension besides `.csv`, the default rule will be used.
15. On the right-hand side, for the **Pass** state configuration, change **State name** to `Pass - Invalid File Ext`. Then, click on the **Output** tab and paste the following into the **Result** textbox:

```
{
  "Error": "InvalidFileFormat"
}
```

16. The **Pass** state is used in a state machine to modify the data that is passed to the next state. In this case, we want to pass a JSON-formatted error message about the file format being invalid to the next state in our pipeline.
17. Click the selector for **Add original input to output using ResultPath** so that that option is selected, and ensure that the dropdown is set to **Combine original input**

with result. In the textbox, enter `$.Payload`.

18. If we receive an `InvalidFileFormat` error, we want to send a notification using the Amazon SNS service. To do so, on the left-hand side, under the **Actions** tab, drag the **Amazon SNS Publish** state to below our **Pass - Invalid File Ext** state.
19. On the right-hand side, on the **Configuration** tab for the **SNS Publish** state, under **API Parameters**, set **Topic** to our previously created SNS topic (`dataeng-failure-notification`). Your state machine should now look as follows:

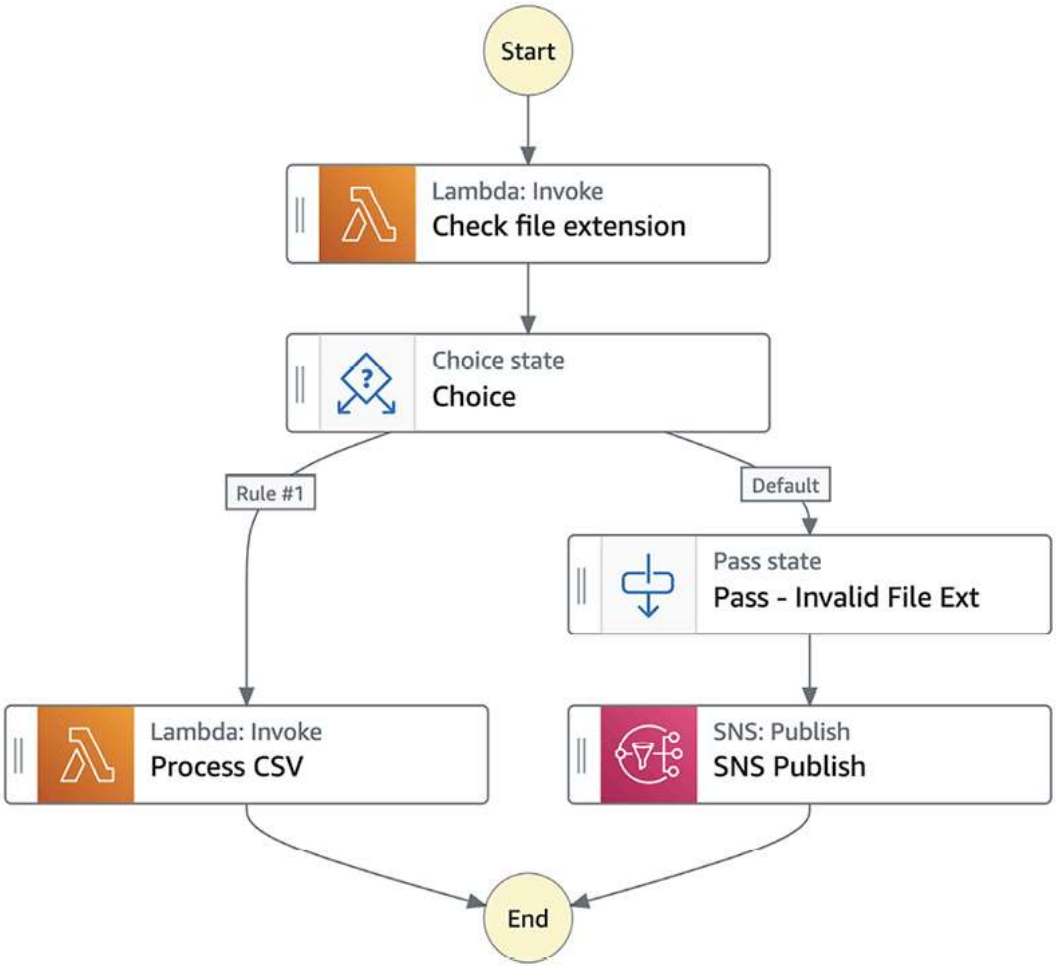

Figure 10.5: The current status of our Step Functions state machine

20. We can now add error handling for our **Process CSV** state. Click on the **Process CSV** state and, on the right-hand side, click on the **Error handling** tab. Under **Catch errors**, click on the + **Add new catcher** button. For **Errors**, select **States.ALL**, for **Fallback state**, select our **SNS Publish** state, and for **result path**, enter `$.Payload`. This configuration means that if our Lambda function fails for any reason (`States.ALL`), we will add the error message to our JSON under a **Payload** key and pass this to our SNS notification state.

21. On the left-hand side, click on the **Flow** tab and drag **Success state** under the **Process CSV** state. Then, drag **Fail state** under the **SNS Publish** state. We do this as we want our Step Functions to show as having failed if, for any reason, something failed and we ended up sending a failure notification using SNS.
    Your finalized state should look as follows:

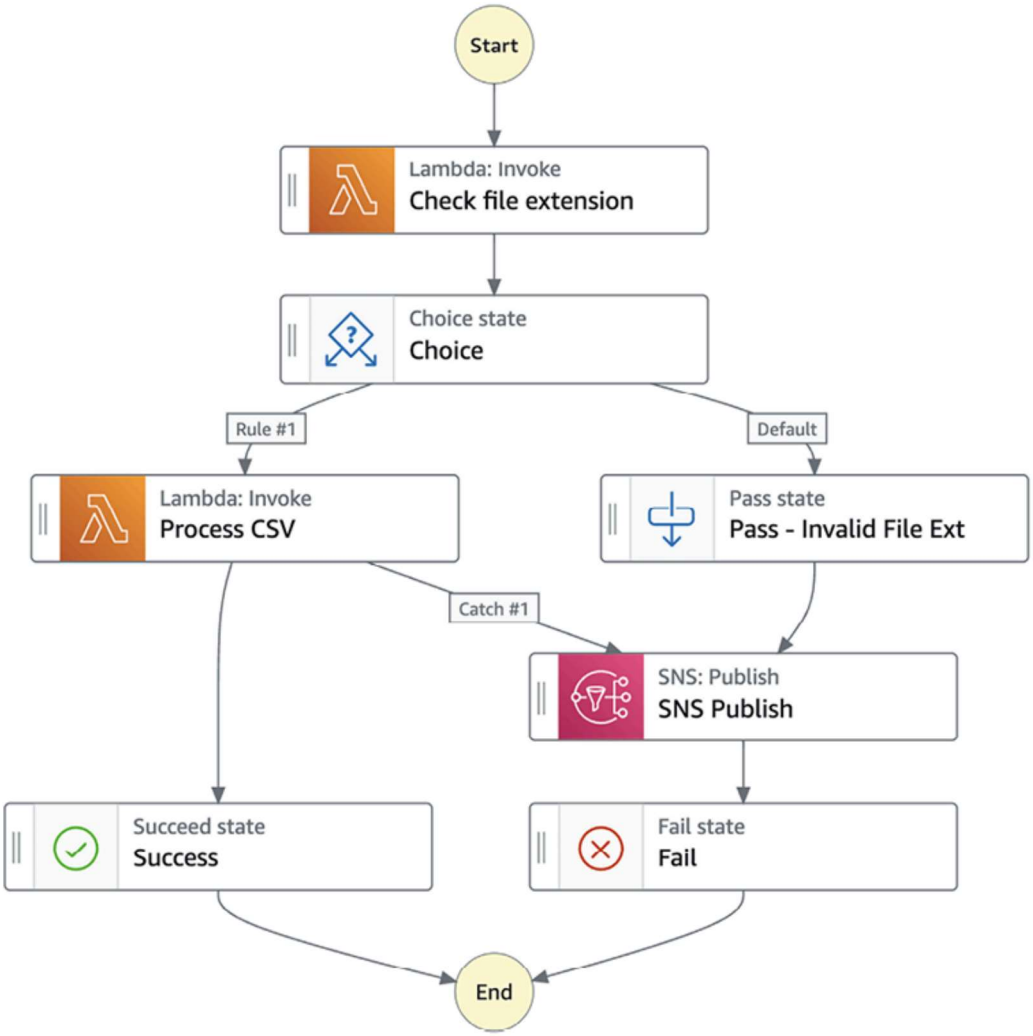

Figure 10.6: The final status of our Step Functions state machine

22. At the top of the screen, click on the **Config** tab.
23. For **State machine name**, enter `ProcessFilesStateMachine` . Leave all the other settings as-is and click **Create** in the top right.
24. On the next screen, click on **Confirm**.

With that, we have created our pipeline orchestration using Step Functions. Now, we want to create an event-driven workflow for triggering our Step Functions state machine. In the next section, we will create a new EventBridge rule that will trigger our state machine whenever a new file is uploaded to a specific S3 bucket.

The **Amazon EventBridge** service is a serverless event bus that can be used to build event-driven workflows. EventBridge can detect events from various AWS services (such as a new file being uploaded to S3) and can be configured to trigger a variety of different targets in response to an event. In our case, we will configure our Step Functions state machine as a target.

Configuring our S3 bucket to send events to EventBridge

In this section, we will configure Amazon EventBridge notifications for our data lake *Clean Zone* bucket. Once this is configured, whenever new files are created in our *Clean Zone* bucket, EventBridge will process the event and trigger our Step Functions state machine.

The following steps will take you through the process of configuring the *Clean Zone* bucket:

1. Navigate to the **Amazon S3 service** at `https://s3.console.aws.amazon.com/s3/home`. Ensure that you are in the Region that you have used for all the exercises in this book.
2. Review the list of buckets, and click on your *Clean Zone* bucket (for example, `dataeng-clean-zone-gse23`).
3. Click on the **Properties** tab for the bucket, scroll down to the **Event Notifications** section, and click on **Edit** next to the **Amazon EventBridge** sub-section.
4. Click the selector for **On**, in order to ensure that notifications are sent to Amazon EventBridge for all events in the bucket. Then, click **Save changes**.

With the above steps, we have configured our *Clean Zone* bucket to send an event containing details about any actions in our S3 bucket (such as new objects created, objects that get read, etc) to EventBridge. In the next section, we will create an EventBridge rule to filter the events, and to send events we are interested in (the creation of new files) to our AWS Step Functions state machine.

Creating an EventBridge rule for triggering our Step Functions state machine

Our final task, before testing our pipeline, is to configure the EventBridge rule that will trigger our Step Functions state machine. Let's get started:

1. Navigate to the **Amazon EventBridge** service at `https://console.aws.amazon.com/events/home`. Ensure that you are in the Region that you have used for all the exercises in this book.

2. From the left-hand panel, click on **Rules**, ensure that the `default` **Event bus** is selected, and then click on **Create rule**.
3. For the rule's name, enter `dataeng-s3-trigger-rule`.
4. For **Rule type**, select **Rule with an event pattern**. Then, click **Next**.
5. For **Event source**, ensure that **AWS events or EventBridge partner events** is selected.
6. Under **Sample event**, for type, ensure that **AWS events** is selected.
7. Under **Sample events**, search for `S3`, and under **Simple Storage Service (S3)**, select **Object created**. This will display an example **S3 Object Created** notification.
8. Under **Creation method**, select **Use pattern form**.
9. Under **Event pattern**, ensure that **AWS services** is selected for **Event source**, and for **AWS service**, search for and select **Simple Storage Service (S3)**. For **Event type**, select **Amazon S3 Event Notification**.
10. Change the selector from **Any event** to **Specific event(s)**. And then from the dropdown, select **Object created**.
11. Change the selector from **Any bucket** to **Specific bucket(s) by name**.
12. In the text box, enter the name of your data lake *Clean Zone* bucket (for example, `dataeng-clean-zone-gse23`). Then click **Next**.
13. Under **Select target(s)**, for **Target 1**, ensure that **AWS service** is selected, and for **Select a target**, search for and choose **Step Functions state machine**.
14. Under **State machine**, select the state machine we created previously (such as `ProcessFileStateMachine`) from the dropdown.
15. Optionally add tags, and then click **Next**.
16. Review the configuration details, and then click **Create rule**.

With the above steps we have created a new **EventBridge** rule that is triggered whenever a new object is created in the S3 bucket we specified. However, we may only want this rule to run when new objects (files) are created in a specific S3 bucket, rather than fsor every new object in the bucket as a whole. We can do this by editing the rule, as follows:

1. Under **Rules**, click on the name of the rule (such as `dataeng-s3-trigger-rule`).
2. Next to **Event pattern**, click on **Edit**.
3. Under the **Event pattern** JSON, click on **Edit pattern**.
4. Modify the JSON to be as follows, but be sure to keep the bucket name that you created (i.e., change `dataeng-clean-zone-initials` to whatever your *Clean Zone* bucket name is):

```
{
  "source": ["aws.s3"],
  "detail-type": ["Object Created"],
  "detail": {
```

```
      "bucket": {
        "name": ["dataeng-clean-zone-initials"]
      },
      "object": {
        "key": [{
          "prefix": "chapter10"
       }]
     }
    }
  }
```

5. Click on **Next**, and then click on **Next** twice more to continue through the screens and accepting the defaults. Then, click on **Update rule**.
   Your completed EventBridge rule should look as follows:

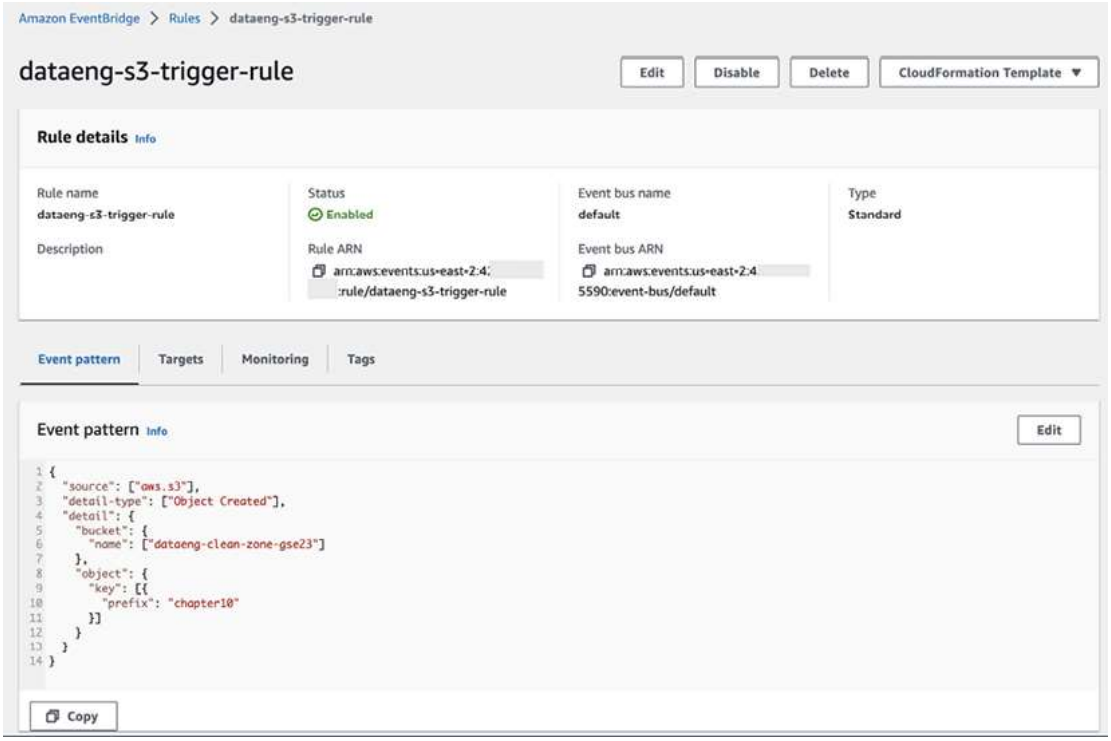

Figure 10.7: Our completed EventBridge rule

With that, we have put together an event-driven workflow to orchestrate a data pipeline using Amazon Step Functions. Our last task is to test our pipeline.

Testing our event-driven data orchestration pipeline

To test our pipeline, we need to upload a file to our *Clean Zone* S3 bucket, into a prefix named `chapter10`. Once the file has been uploaded, the rule we created in Amazon EventBridge will cause our Step Functions state machine to be triggered:

1. Navigate to the Amazon S3 service at `https://s3.console.aws.amazon.com/s3`.
2. From the list of buckets, click on the **dataeng-clean-zone-<initials>** bucket.
3. Click on **Create folder** to create a new folder. For **Folder name**, specify `chapter10`. Then, click **Create folder**.
4. Click on the new folder (`chapter10`) to move into that folder.
5. Click on **Upload**, and then **Add files**. Browse your computer for a file with a CSV extension (if you cannot find one, create a new, empty file and make sure to save it with an extension of `.csv`).
6. Leave the other settings as-is and click **Upload**.
7. Navigate to the **AWS Step Functions** service at `https://console.aws.amazon.com/states`.
8. Click on the state machine we created earlier (`ProcessFilesStateMachine`). From the list of **Executions**, see whether the state machine **Succeeded** or **Failed**. Click on the **Name** property of the execution for more details.
   Note that you may see two executions, with the first execution failing and the second one succeeding (depending on the random number generator result). The first execution is triggered by the creation of the new folder/prefix (`chapter10`), and this fails with an invalid file extension failure, as the prefix obviously does not have a `.csv` extension. The second execution (where you uploaded the file with a .CSV extension) will either succeed or fail, depending on what random number was generated by our second Lambda function.

9. Reupload the same .csv file multiple times and notice how some executions succeed and some fail. The random number generator has a 66% chance of generating the number 1 or 2 and a 33% chance of generating the number 0. When the number 0 is generated, the function will fail, so throughout many executions, approximately one-third should fail.
   The following diagram shows an example of what our state machine looks like after an execution where 0 was generated as a random number, causing the Lambda function to fail:

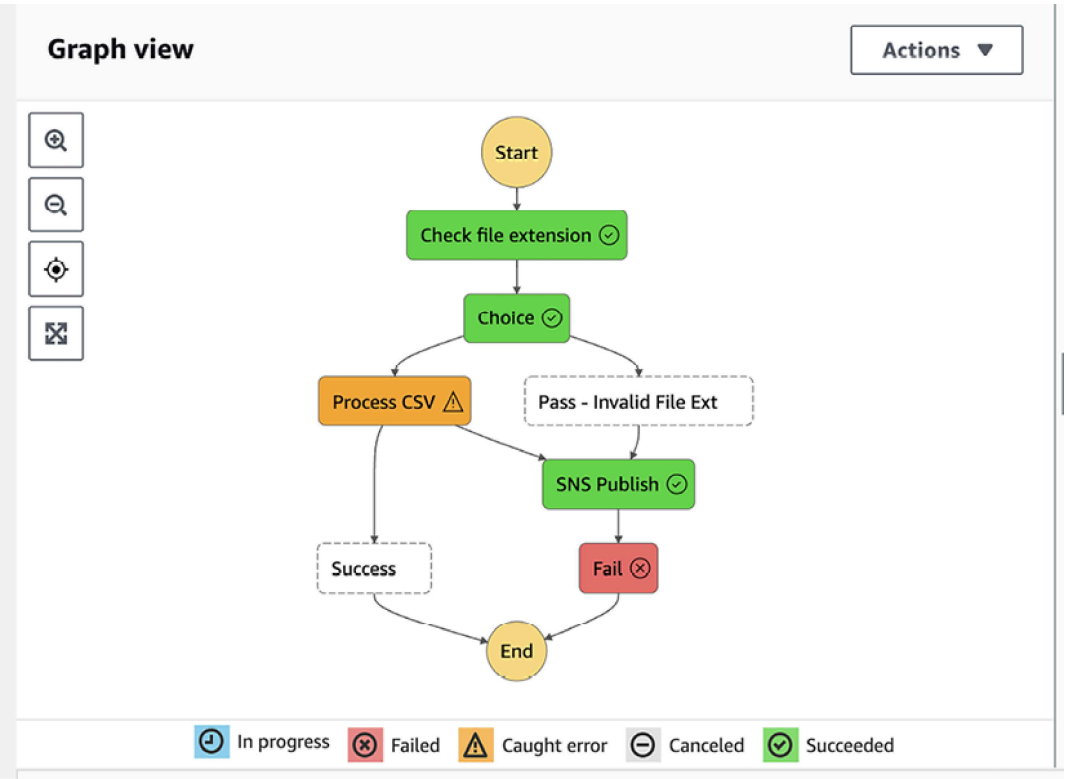

Figure 10.8: An example of a state machine run when the random number generator generated a 0, resulting in a failed state machine

10. After a failed execution, check the email address that you specified when you configured the Amazon SNS notification service. If you previously confirmed your SNS subscription, you should receive an email each time the state machine fails with details on the error (such as `"Error":"ZeroDivisionError"`).
11. Now, upload another file to the same Amazon S3 bucket, but ensure that this file has an extension other than .csv (for example, `.pdf`). When you're viewing the execution details for your state machine, you should see that the choice state proceeded to the **Pass – Invalid File Ext** state and then also published an SNS notification to your email. The error listed in the email should be `"Error":"InvalidFileFormat"`.

In the hands-on activity for this chapter, we created a serverless pipeline that we orchestrated using the AWS Step Functions service. Our pipeline was configured to be event-driven via the Amazon EventBridge service, which let us trigger the pipeline in response to a new file being uploaded to a specific prefix in our Amazon S3 *Clean Zone* bucket.

You could easily modify this state machine to handle different types of files, in different ways. For example, you could create a Lambda function that converts a CSV file to Parquet format, but that passes an image file (those with JPEG or PNG extensions) to a different Lambda function that creates a thumbnail of the image.

This was a fairly simple example of a data pipeline. However, AWS Step Functions can be used to orchestrate far more complex data pipelines, with advanced error handling and retries. For more information on advanced error handling, see the AWS blog titled *Handling Errors, Retries, and Adding Alerting to Step Functions State Machine Executions* (https://aws.amazon.com/blogs/developer/handling-errors-retries-and-adding-alerting-to-step-function-state-machine-executions/).

Summary

In this chapter, we looked at a critical part of a data engineer's job–designing and orchestrating data pipelines. First, we examined some of the core concepts around data pipelines, such as scheduled and event-based pipelines, and how to handle failures and retries.

We then looked at four different AWS services that can be used for creating and orchestrating data pipelines. This included AWS Data Pipeline (now in maintenance mode), AWS Glue workflows, Amazon MWAA, and AWS Step Functions.

Then, in the hands-on section of this chapter, we built an event-driven pipeline. We used two AWS Lambda functions for processing, and an Amazon SNS topic for sending out notifications about failures. Then, we put these pieces of our data pipeline together into a state machine orchestrated by AWS Step Functions. We also looked at how to handle errors.

So far, we have looked at how to design the high-level architecture for a data pipeline and examined services for ingesting, transforming, and consuming data. In this chapter, we put some of these concepts together in the form of an orchestrated data pipeline.

In the remaining chapters of this book, we will take a deeper dive into some of the services for data consumption, including services for ad hoc SQL queries, services for data visualization, as well as an overview of machine learning and Artificial Intelligence services for drawing additional insights from our data.

In the next chapter, we will do a deeper dive into the Amazon Athena service, which is used for ad hoc data exploration, using the power of SQL.

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