16
Building a Modern Data Platform on AWS

As we near the end of this book, we will review high-level concepts around building a modern data platform on AWS. We could easily devote another whole book to this topic alone, but in this chapter, we will provide at least an overview to give you a strong foundation on how to approach the build-out of a modern data platform.

There are many different pieces to the puzzle of building a modern data platform, and this chapter will build on many of the other topics we have covered in this book (such as data meshes and modern table formats) alongside introducing topics we have not yet covered (such as Agile development and CI/CD pipelines).

The goal of this chapter is to help you think through how to bring together many of the different concepts you have learned in this book to create a data platform that supports both the data producers and data consumers in your organization. This chapter is not a complete guide to building a data platform, but rather introduces important concepts and tools to enable you to start planning the building of your own modern data platform on AWS.

In this chapter, we will cover the following topics:

- Goals of a modern data platform
- Deciding whether to build or buy your data platform
- DataOps as an approach to building data platforms
- Hands-on – automated deployment of data platform components, and data transformation code

Before we get started, review the following *Technical requirements* section. This lists the prerequisites for performing the hands-on activity at the end of this chapter.

Technical requirements

In the last section of this chapter, we will go through a hands-on exercise that automates the deployment of components that could be used in a data platform, as well as the code for a data engineering pipeline. This will require permission for services such as AWS CloudFormation, AWS CodeCommit, AWS CodeDeploy, as well as AWS Glue and various other services. As with the other hands-on activities in this book, having access to an administrator user in your AWS account should give you the permissions needed to complete these activities.

You can access more information about running the exercises in this chapter using the following link: https://github.com/PacktPublishing/Data-Engineering-with-AWS-2nd-edition/tree/main/Chapter16

Goals of a modern data platform

In *Chapter 15, Implementing a Data Mesh Strategy*, we discussed how a central data platform team is responsible for building a platform that makes it easy for both data producers and data consumers to work with organizational data.

A data platform is intended to provide a system where multiple teams from across an organization can easily ingest data (including both structured and semi-structured, via batch and streaming), process the ingested data, and create new data products by joining datasets. It should also provide data governance controls, a catalog for making data discoverable across the organization, and the ability to easily share datasets across different teams/data domains.

Let's review some of the top goals for a modern data platform, after which we will explore approaches to building these data platforms.

A flexible and agile platform

As we all know, the only constant is change. We have seen this throughout this book as we spoke about new table formats, including Apache Iceberg, and new data architecture approaches such as the data mesh approach. AWS services also constantly evolve, and new services are introduced (both by AWS and other vendors). In this second edition of the book, there are three new chapters (including this one), as well as countless updates related to new AWS services and features (such as EMR Serverless, Bedrock, Redshift support for Iceberg, Glue support for Ray.io, DataZone, and more).

As a result, it is critical when building a data platform to ensure that the platform is flexible enough to incorporate new technologies and approaches. And not only should the platform be adaptable, but also agile enough to adapt and incorporate changes quickly. This requires an agile approach to the development of the platform, as we discuss later in this chapter.

A scalable platform

When building a new platform, you want to be able to start with a minimally viable product (MVP), or minimally loveable product (MLP), as some people prefer to call it. This means that you build a working platform with a minimal set of critical features, and then over time you add additional features. It also means that you do a staged roll-out of the platform, starting with perhaps a single domain, and then onboarding additional domains over time.

This means that right from the start, you need to build a platform that is scalable. On day one of the platform being live, you may have just one or two datasets of a few hundred GBs. But as you onboard additional domains/lines of business, the data volumes may grow to tens, and then hundreds, of TBs.

One of the big advantages of building a modern data platform on the cloud is the built-in scalability. For example, an Amazon S3 bucket can be used to store hundreds of GBs of data to start with, and yet without any additional configuration, the bucket can scale to store hundreds of TBs of data. The same goes for most other AWS services, specifically those that are serverless. For example, Amazon Kinesis in on-demand mode is able to automatically scale in response to changing data traffic.

As a result, one of the key enablers of a data platform that can easily scale is the use of AWS serverless services.

A well-governed platform

Another key attribute for a modern data platform is that it must support good governance of all data that is stored and processed on the platform.

In *Chapter 15*, *Implementing a Data Mesh Strategy*, we discussed how one of the key principals of a data mesh is *federated computational governance*. In summary, a central governance team, made up of representatives from each of the domains, works together to decide on the minimal required governance standards for the platform (such as compliance with data governance regulations, etc.). Each domain must then comply with those governance standards, but they may also implement their own additional governance requirements for their specific domain.

The central data platform needs to be able to support both the central common data governance requirements, and the governance requirements for each individual domain. This means that strong data governance functionality needs to be a core part of the platform from the start. Good data governance controls must be considered part of the minimally viable/lovable product that we spoke about earlier.

A secure platform

Another obvious but key requirement is that the data platform must be secure. This is an aspect of data governance generally, but since it is so critical, we call it out separately here.

Security on the platform needs to cover things such as ensuring that all data both at rest and in transit is encrypted. In addition, security also involves access control, making sure that only authorized users have access to data on the data platform, and then only the level of access to the data they need as opposed to system-wide data access. Another aspect of security is ensuring that data access is logged, so that access attempts can be audited by the security team.

In an overlap with data governance, security may also include using a service such as **Amazon Macie** to identify whether any PII information is contained in data files so that it can be secured if so.

An easy-to-use, self-serve platform

Even if you get all the previous goals correct – you have an agile, scalable, well-governed, and secure platform – but that platform is difficult to use, then its success will always be limited and it is likely to fail.

It is critical that different domains/lines of business are able to easily onboard onto the platform, and that users can quickly become productive with producing and/or consuming data. This requires both organizational effort (ensuring that you have a team that helps domains onboard to the platform), and making sure that use of the platform is well documented and the platform is easy to use from a technical standpoint. The platform should also support self-service – that is, once onboarded, teams should be able to add new data products, subscribe to data products from other teams, and perform most functions without needing to depend on a central data platform team to perform certain tasks.

The goal is to have the tools available that make it easy to ingest data into the platform, transform that data, apply required governance controls, register the data in a central catalog, and then enable data consumers to search for, subscribe to, and consume data products from across the organization. All of this should be doable without needing to raise tickets or request help from the central data platform team.

Having reviewed some of the key goals for central data platforms, let's now look at a critical decision point – whether to build a data platform or buy one.

Deciding whether to build or buy a data platform

The question of whether to build or buy applies to many different purchasing decisions that an organization needs to make. Some of these decisions are pretty obvious – for example, not many organizations will choose to build their own power plant and generate their own power, rather than just purchasing power from their local utility company.

Within the IT realm, there is likely to be a mix of building and buying, depending on the size of the organization. For example, most organizations that need systems for HR, Customer Relationship Management (CRM), or Enterprise Resource Planning (ERP) will purchase these from one of the many vendors that have built these products for many years. However, many organizations will choose to build and manage their own website and mobile app, including the microservices that power those systems.

Organizations also have a choice when it comes to their approach to implementing a modern data platform. However, there are a number of factors that you need to take into consideration when deciding whether to build or buy a data platform. Let's start by looking at what may be involved when you choose to buy a platform.

Choosing to buy a data platform

One option is to purchase a unified data platform that provides the processing and query engine and most of the components an organization needs, and then integrate other components as required (such as BI tools or database ingestion tools). Many vendors provide data platform products, however, in this section, we will primarily reference two of the most common data platforms – Databricks and Snowflake.

Databricks offers the **Databricks Lakehouse Platform**, designed to "*provide all the components needed to unify data, analytics, and AI.*" **Snowflake** offers the **Snowflake Platform**, designed to "*connect businesses globally, across any type of scale of data and many different workloads.*"

Depending on the vendor and platform you select, most data platforms that you purchase include the following components:

- **Data storage**: When you ingest data into most vendor-provided data platforms, the data is converted into an optimized format. This may be in a proprietary format, such as the compressed, columnar format that Snowflake stores data in, or an open format, such as the Delta Lake format that Databricks uses to store and manage data tables. Modern data platforms mostly make use of object storage such as Amazon S3 to physically store files.
- **Data transformation engine**: A core part of these platforms is an engine that enables you to create ETL/ELT jobs to transform your data, applying business logic across multiple datasets. The most popular languages for performing data transformation are SQL and Python (including PySpark, a Python API for Apache Spark). Databricks provides the ability to transform data using either Apache Spark or SQL, while Snowflake primarily provides a SQL-based interface.
- **Data query editor**: Another key component of data platforms is a query editor that enables data producers and consumers to interact with the data on the platform. This may be a SQL-based editor for running SQL queries against data, or something different, such as a notebook interface that enables users to work with platform data using code. Both Databricks and Snowflake provide a rich SQL query editor that also enables the ability to visualize query results (i.e., viewing the results as a bar chart, line chart, heat grid, etc.).

- **Data services**: Each platform will provide a number of services that you can use to manage and monitor your data platform, implement data governance, etc. This may include a data catalog for managing metadata related to your datasets (such as the Databricks Unity Catalog), as well as services that enable data sharing within the data platform and across different teams (such as Snowflake Secure Data Sharing, or Databricks Delta Share). Advanced services may also be included, such as the ability to securely combine data with a partner's dataset in a way that protects data privacy and limits access to the underlying granular data (such as with the clean room solutions offered by both Databricks and Snowflake).

Note that even when you purchase a platform from a vendor, you are still likely to need to integrate other third-party services and applications with your data platform. For example, you may have an application that streams data via Kinesis Data Streams or Amazon MSK (a managed streaming Kafka service), and you need to integrate that into your data platform. You may also want to use a commercial tool that enables easy data ingestion from many different data sources (such as Upsolver, Striim, Matillion, Fivetran, and others).

On the data consumption side, you are also likely to need to integrate a BI visualization tool such as QuickSight or Tableau.

When to buy a data platform

The primary benefit of purchasing a data platform from a vendor is that they provide and integrate all the common components that are needed for a modern data platform. This makes the implementation of the platform much simpler and reduces the skillsets required by the organization to implement and manage the platform. For organizations without existing engineering skills, this is a significant benefit.

However, when you purchase a data platform from a vendor, you are limited to the functionality that the vendor provides. As a result, it is important to fully evaluate different vendor offerings to ensure that the platform meets your needs and is likely to continue to do so as your requirements change over time. Once an investment has been made in a specific vendor's data platform, it can be very difficult to change to a different platform, and therefore it is critical that you evaluate your vendor's ability to support your organization over a 5-10 year timeframe.

Purchasing a platform is also often more expensive than using equivalent cloud-native services. The data platform vendors provide the integration between components managing the infrastructure and software for you and provide support when you run into issues. However, this often comes at a premium cost. Therefore it is important that you understand all the factors that can influence the cost of the vendor solution (for example, some vendors charge an additional premium for more advanced features) and project what your costs are likely to be over a longer time period, based on your expected data volume growth and additional features you may need over time.

Another benefit of purchasing a data platform is that, often, these platforms are supported across multiple clouds. For many organizations, it is simpler to standardize, and build on, a single cloud, but for large organizations, different teams may have elected over time to use different cloud providers. If your organization already has teams building in different clouds, then the ability to have a single data platform (such as Databricks or Snowflake) that can be run in AWS, Azure, and GCP may be a significant benefit.

Let's now look at some of the reasons that it may make sense to build a platform using cloud-native services and third-party tools, instead of purchasing a platform from a vendor.

Choosing to build a data platform

An alternative to buying a data platform from a vendor is to build your own platform using cloud-native services. Throughout this book, we have discussed a number of AWS services that can be integrated together to build a data platform:

- **Data storage**: Most modern data platforms are built using cloud object stores, and within AWS you can build your data platform with Amazon S3 as the physical storage layer. As we discussed in *Chapter 14*, *Building Transactional Data Lakes*, you can also use one of the modern open-table formats to store your data in an optimized format, such as Apache Iceberg. For smaller projects and environments where the primary skillset is SQL, you may decide to store your data in Amazon Redshift (although with Redshift RA3 nodes, the persistence layer does use Amazon S3 under the hood).
- **Data transformation engine**: We have discussed a number of AWS services that can be used to transform data as part of an ETL or ELT solution. For example, you can transform data using Apache Spark with the AWS Glue or Amazon EMR services, or use Python and pandas for light data transformation tasks using AWS Lambda. Alternatively, you can use SQL to transform your data using Amazon Athena or Amazon Redshift.
- **Data query editor**: If you primary storage platform is Redshift, you can use Redshift Query Editor for working with your data. Alternatively, if using Amazon S3 to store your data, you can use Amazon Athena as your query editor.
- **Data services**: AWS has a wide range of analytic services that can be integrated as part of your data platform. For example, AWS Lake Formation and Amazon DataZone can be used as a technical and business catalog respectively, and Lake Formation can also be used to manage access control and for sharing data across accounts. The AWS Clean Rooms service can be used to share data in S3 with partners in a secure, privacy-protecting way, and AWS Data Exchange can be used to directly access data from third-party data suppliers.

While building a data platform using cloud-native components is not the appropriate choice for all organizations, it does carry a number of advantages over buying one, as we discuss next.

When to build a data platform

One of the primary advantages of building a data platform is that you have a lot of flexibility in which components you use, and cloud-native services are often available at a lower cost than buying a data platform solution from a vendor.

When you build a data platform, you get to customize the platform based on your specific business requirements and have a wider variety of tools that you can integrate with the platform to meet the requirements of different teams within your organization. You can also more easily swap out different components of the platform over time than if you had purchased a full data platform solution from a single vendor.

In most cases, you will need some level of data engineering skills regardless of whether you use a vendor-provided data platform or a platform that you build yourself. These data engineers need to apply business logic to datasets in order to build analytical data products, which can be done through SQL or code-based transforms. Your data engineers also need to integrate the platform with other components, such as data ingestion services and BI visualization tools. However, when building a platform, your engineering team will also be responsible for creat-

ing integrations between the various AWS components that make up the core data platform (such as AWS Glue, Lambda, Lake Formation, DataZone, etc.).

Companies that have generalist developers and DevOps skills are well suited to building their own data platform. This is true for both small start-up type companies and large enterprises. In small start-ups, where budgets are tight, engineers can build their own data platform and start small with very low costs, and then scale up over time as the business grows. For large enterprises, there is often already a mix of different analytic tools used by different lines of business, and it may be easier to integrate these different technologies into a custom-built modern data platform than it would be to force all lines of business to standardize and migrate to a solution from a single vendor.

However, for mid-size companies that do not have developers and DevOps-type resources, it may make more sense for them to purchase a data platform solution from a vendor, and have the vendor (or a partner recommended by the vendor) implement and integrate the solution for them.

A third way – implementing an open-source data platform

As somewhat of a compromise between buying a vendor-provided data platform and building one from scratch, another option is to implement an open-source data platform. This is closer to building a data platform, as implementing most open-source data platforms requires downloading the source code from a repository such as GitHub, and then having engineering teams implement and customize the solution. Your engineering teams will also need to integrate your different data sources with the platform, as most open-source data platforms do not have connectors for all common sources.

However, it does give the engineering team a headstart and provides a mature platform with many features. In addition, it allows the engineering team to customize the platform based on the requirements of the business.

In the following section, we provide a brief overview of a data platform created and open-sourced by AWS Professional Services.

The Serverless Data Lake Framework (SDLF)

The **Serverless Data Lake Framework (SDLF)** is an open-source project that provides a data platform that accelerates the delivery of enterprise data lakes on AWS. This includes a number of production-ready best-practice templates that speed up the process of implementing data pipelines on AWS. This framework has been implemented at a number of large organizations, including Formula 1 motorsports racing, Amazon retail in Ireland, and Naranja Finance in Argentina (amongst many other companies).

Some of the best practices that are implemented via the SDLF include:

- **Infrastructure-as-Code and version control**: The base SDLF implementation, along with customizations and data transformation tasks, are all managed through a CI/CD pipeline and a code repository. The SDLF avoids the need to do any manual implementation via the AWS console, instead deploying infrastructure and code via DevOps pipelines. This also means that any changes to the data platform or data transformation pipelines are managed via a code repository, providing version control.
- **Scalability using serverless technologies**: The SDLF makes extensive use of serverless technologies for the core of the data platform. Serverless services can easily scale from low data volumes to handling much larger data volumes, and are often the most cost-effective

approach as you only pay for resources when tasks are running (you never pay for idle time).
- **Built-in monitoring and alerting**: The SDLF data platform includes built-in monitoring (using the ELK stack) and alerting (via CloudWatch alarms). This ensures that issues can be detected quickly and that logs are easily accessible for troubleshooting issues.

Deploying and managing the SDLF does require DevOps skills, so is not intended for environments that do not have relevant engineering skills. Deploying the SDLF should not be viewed as an alternative to purchasing a data platform from a vendor like Databricks or Snowflake; however, if you are considering building a data platform, then the SDLF can help accelerate the data platform build while ensuring many best practices are applied.

In this next section, we will look at some of the key components of the SDLF.

Core SDLF concepts

There are various layers that make up the SDLF, starting from the foundational layer that provides the core data platform functionality. Once the **foundations** have been deployed, **teams** are able to build **pipelines** (which control the **transformation** processes that are applied to a **dataset**). Let's look into each of these terms in more detail.

Foundations

When you deploy the SDLF foundation layer, you are deploying components to be used by all teams and all pipelines. This includes components such as Amazon S3 buckets for storage, DynamoDB tables (for configuration and the metadata catalog that tracks pipeline executions), and the ELK stack for monitoring.

Teams

The team layer deploys resources for a specific team (that is, a group of people that work together, such as a specific line of business, or a smaller team within a line of business). A team develops and deploys datasets, transformations, pipelines, and code repositories that are unique to that specific team.

Datasets

With the SDLF, a dataset is a logical grouping of data – this could be a single table, or an entire database consisting of multiple tables. For example, a dataset could constitute the tables imported from a relational database, or a group of files coming from a streaming data source (such as IoT data from a factory system).

Pipelines

A pipeline is a logical view of an ETL process, laying out the steps that data goes through as it is transformed. Teams create pipelines, and may create multiple pipelines for different datasets.

With the SDLF, each pipeline is split into multiple stages, most commonly Stage A (which is intended for doing an initial light transform on a single file), and Stage B (which is intended for heavier transforms, such as joining datasets or applying business logic).

The orchestration of these pipelines is done using AWS Step Functions. Each stage has a Step Functions workflow that defines the transformation steps that will be completed as part of the pipeline.

Transformations

Transformations perform the data transformation tasks – converting files to Parquet format, joining datasets, aggregating columns, etc. These transformations are run by the different stages of the pipeline. For example, a pipeline may consist of a Step Functions workflow that runs a Glue job to apply business logic to data, and then runs a Glue crawler to crawl the new data and add it to the Glue Data Catalog. The transformation would be the Spark code that the Glue job runs.

To learn more about the **Serverless Data Lake Framework** (**SDLF**), use the following links:

- **SDLF GitHub page**: `https://github.com/awslabs/aws-serverless-data-lake-framework`
- **SDLF documentation**: `https://sdlf.readthedocs.io/en/latest/index.html`

Having looked at options for buying, building, or implementing an open-source solution for a data platform, let's now review how you can use a DataOps approach to building and maintaining your data platform and transformation code.

DataOps as an approach to building data platforms

DataOps is a term that has been around since at least 2015 and refers to an agile approach to building data platforms and data products that borrows from some of the concepts of DevOps. Where DevOps transformed the approach to how software is engineered, DataOps transforms the approach by which data products are built.

Much like it is difficult to give an exact definition or outline an exact approach for other concepts we have discussed (such as data lakes and data meshes), it is similarly difficult to tie down one clear-cut definition for what is meant by DataOps. The original author of the term may have had a clear definition of what they meant, but over time, the term may come to mean different things to different people, and the meaning of the term as a whole may evolve.

In this section, we will attempt to focus on some of the core concepts of DataOps, and specifically, how they apply to building a data platform and data products. There is more to DataOps than we can cover in a single chapter, so in this section, we will focus on two of the key aspects – automation and observability.

Automation and observability as a key for DataOps

While there may be different definitions of what DataOps is, at its core it is about automation and observability. This includes automating the deployment of the data platform (and updates to the platform), as well as deployment and updates to the transform code that produces data products.

A key part of this automation is managing infrastructure for the data platform and logic for data transformations as version-managed code.

In addition, there should be automation for monitoring and alerting for the data platform and data transformations. If the platform is designed to trigger an AWS Lambda job to run in response to a specific event (such as a new data file being received) and the Lambda function fails to execute, a data platform engineer should receive an alert. In the same way, if the code within a Lambda function causes an error and the function fails, a data engineer should be alerted.

There should also be observability dashboards that enable data platform owners and data engineers to view key metrics related to their area of responsibility. A platform engineer should easily be able to view the throughput of Kinesis Data Streams or the number of executions of a Lambda function, and a data engineer should be able to easily view the status of a specific pipeline they have engineered. When there is a failure, engineers should be able to easily access relevant log files in order to troubleshoot and identify the root cause of the issue.

Another key success factor for DataOps is ensuring that teams work with an **Agile approach**, working in short sprints in order to constantly evolve and enhance the data platform and products, rather than planning months-long release cycles. To learn more about the Agile approach to software development, see `https://www.atlassian.com/agile`.

Let's take a closer look at the topic of automating infrastructure and code deployment.

Automating infrastructure and code deployment

When you deploy data platform infrastructure (such as a Snowflake or Redshift cluster, a Databricks cluster or an AWS Glue job, or a Kinesis Data Stream) the deployment should **not** be done manually using the console or by typing commands on the command line, but should be automated using a provisioning tool. Two common tools for this purpose are **Terraform** (by HashiCorp) and **CloudFormation** (an AWS service).

In the same way, when deploying data transformation code (such as the code for a Glue job or Databricks job, or a SQL transform in Snowflake or Redshift), this code should be managed and deployed from a version-controlled code repository. Common tools for this include **GitHub**, **GitLab**, **AWS CodeCommit**, **Bitbucket**, and **Azure DevOps**.

When developing code, developers should build unit tests (among other types of tests), and as part of the automation of code deployment, the pipeline should use these tests to validate the code. It is also recommended to perform security scans of code as part of the pipeline and to implement other software development best practices.

While it may be argued that managing the code for these deployments via these tools may add additional overhead to the deployment process, it significantly increases the reliability and security of the deployment and improves the reliability of making updates to either infrastructure or data transformation code. If an update does fail, then having all infrastructure, pipeline, and transformation code managed in a version control system means that rolling back to the previous version of the code can be done quickly.

For example, let's say someone deploys some data transformation code in a Glue job by directly writing the code for the Glue job in the AWS console. When that code needs to be updated, another engineer may use the AWS Management Console to modify the code. With this approach, no history of the code changes will be stored and an engineer can change the code without anyone else needing to approve those changes.

In comparison, if the code for the Glue job is managed via a code management solution (such as AWS CodeCommit or GitHub), then each version of the code is stored and available to review. When code is updated, you can build in approval requirements which require the code to be reviewed by other members of the team, and perhaps a security engineer, before the code is deployed via an automated pipeline. In addition, code can be developed and tested in a development environment and then promoted to be deployed in the QA or production environment via a pipeline.

Let's now look at the importance of observability for data platforms and data engineering pipelines.

Automating observability

The other key part of building data platforms and products is to ensure that there is clear observability – this means that teams can monitor the current status of operations, identify trends that indicate when things are starting to go wrong, and deep-dive to find the root cause of issues when things do go wrong.

There are three primary tools that enable this:

- **Dashboards**: Having dashboards that are constantly updated enables a team to get an accurate view of current operations. By looking at various dashboards, teams can view Key Performance Indicators (KPIs) that show the health of a system. For example, a platform engineer can view a **CloudWatch dashboard** for **Simple Queue Service (SQS)** to determine whether the queue size is growing, indicating an issue with processing the queue. In a similar way, a data engineer can view a list of all DAGs in **Apache Airflow** and easily see when a specific DAG last ran, the recent tasks for a DAG, the number of runs, etc.
- **Alerts**: The platform should have a way of sending automated alerts when things are not operating normally or if failures are detected. For example, **CloudWatch alarms** can be configured to send an email or text alert if a queue grows beyond a certain threshold, and **Airflow Service Level Agreements (SLAs)** can be used to generate an email if the expected time it takes for a specific task to complete exceeds an agreed length of time.
- **Logs**: The other tool that teams need is an effective log collection system that includes functionality for doing advanced searches against logs. When things do go wrong (as they will!), teams need an effective way to search through logs to help troubleshoot the issue. For example, **CloudWatch Logs** or **Amazon OpenSearch** can be used to store logs from the various systems across a data platform and engineers can easily search through the logs to troubleshoot failures.

To learn more about DataOps, you can access a free, on-demand DataOps certification course titled *The Fundamentals of DataOps*. This course, along with a certification assessment, is offered by *DataKitchen*, a commercial provider of DataOps solutions. Access the course at `https://info.datakitchen.io/training-certification-dataops-fundamentals`.

Having briefly discussed a DataOps approach to developing data platforms and data products, let's now look at some of the core AWS services that can be used to implement a DataOps approach.

AWS services for implementing a DataOps approach

In this section, we'll explore a quick overview of some of the core AWS services that can help you implement a DataOps approach to building your data platform and data products. Then, in the hands-on section of this chapter, you can follow along to get some practical experience by putting some of these services into action.

AWS services for infrastructure deployment

There are two primary tools within AWS that enable you to manage and deploy your **Infrastructure as Code (IaC)**, instead of manually deploying resources via the AWS console or the command line.

AWS CloudFormation

AWS CloudFormation enables you to define your AWS resources within a template (in either YAML or JSON format), and then automatically deploy those resources into your AWS account. When you need to modify some aspect of the infrastructure, you can modify and redeploy the template and CloudFormation will intelligently update the resources.

For example, you can define a simple AWS Glue job using the following YAML template:

```
AWSTemplateFormatVersion: '2010-09-09'
# Sample template to define an AWS Glue job (in YAML format)
Resources:
# The script file already exists in S3 and is called by this job
  GlueSampleJob:
    Type: AWS::Glue::Job
    Properties:
      Role: DataEngGlueCWS3CuratedZoneRole
      Description: Glue job to denormalize film and category tables
      Command:
        Name: glueetl
        ScriptLocation: "s3://aws-glue-assets-1234567890-us-east-2/scripts/Film Category Denormalization.py
      WorkerType: G.1X
      NumberOfWorkers: 2
      GlueVersion: "3.0"
      Name: Film Category Denormalization via CFN
```

The above template deploys a Glue job to run the Film Category Denormalization job (which we created in an earlier chapter using the Glue Studio console).

CloudFormation also supports the ability to use parameters in templates, and at deployment time the parameter values can be passed to CloudFormation. This enables you to use the same template to deploy resources into different environments, such as your dev, test, and production environments. For example, the `ScriptLocation` could be referenced as a parameter, and then at the time of deployment the correct S3 bucket name for the specific environment could be passed in. Or the `WorkerType` and `NumberOfWorkers` could be parameters, so that you deploy smaller resources in the dev and test environments, and more powerful resources for the production job deployment.

If you are building a data platform using all AWS services, then **CloudFormation** is a good option for automating your infrastructure deployments and updates. However, if you have a multi-cloud approach, or you also want to automate the deployment of other non-AWS services (such as Snowflake or Databricks), then **Terraform** (a product from HashiCorp) provides similar functionality for both AWS and non-AWS resources. For more information on Terraform, see **https://www.terraform.io/**.

Let's now look at another related service that enables us to create CloudFormation templates with recommended best practices via code.

AWS Cloud Development Kit (CDK)

The AWS CDK enables you to define resources using popular programming languages including TypeScript, JavaScript, Python, Java, C#/.Net, and Go. You define the resources in your code and the CDK then creates a CloudFormation template that incorporates best practices for the specific type of resource that you are creating.

Some of the benefits of using the AWS CDK to define your AWS resources as code are as follows:

- **It allows you to specify high-level constructs in your code, and have CDK automatically create CloudFormation templates that provide best practice and contain secure defaults:**
  This enables you to define resources that follow best practices, with less code. For example, the AWS documentation demonstrates how just 13 lines of Python code that provides high-level constructs can be used to create a CloudFormation template of over 500 lines. See: `https://docs.aws.amazon.com/cdk/v2/guide/home.html`.

- **It allows you to use common programming approaches to model the design of your system:**
  For example, you can use the AWS CDK to write Python code that defines AWS resources that make up your data platform, and make use of programming constructs such as parameters, if … then … else conditions, loops, and more.

- **It enables the import of existing CloudFormation templates for use in your AWS CDK code:**
  With this approach, you can migrate any existing CloudFormation template to the AWS CDK one piece at a time. See the AWS documentation on this at `https://docs.aws.amazon.com/cdk/v2/guide/use_cfn_template.html`.

- **It enables you to centralize the code that defines your data platform resources alongside the code for data transformation jobs, CI/CD pipelines, etc.:**
  Source Control Management (SCM) systems such as Git (which we will cover in more detail in the next section) enable you to securely manage your code in a centralized location and include features that simplify collaboration and help manage conflicts. By defining your resources using the AWS CDK, you can manage all your code in central repositories.

AWS has a blog post that demonstrates how to use the AWS CDK to create a continuous integration and continuous deployment (CI/CD) pipeline to define, test, provision, and manage changes to an AWS Glue-based pipeline. We will not cover the AWS CDK in any more detail here, but refer to the following AWS blog post to learn more about how the CDK can be used with data pipelines: `https://aws.amazon.com/blogs/big-data/build-test-and-deploy-etl-solutions-using-aws-glue-and-aws-cdk-based-ci-cd-pipelines/`.

Let's now look at AWS services for managing and deploying code.

AWS code management and deployment services

Source Control Management (SCM) systems enable you to store code (along with documentation and other files) in a way that each version of a file is stored and tracked. It also helps manage potential conflicts when you have multiple engineers working to update the same set of files. This has many benefits, including:

- The ability to easily revert to a previous version of code
- Being able to check which person on the team made a change to a specific part of the code
- The ability for multiple people to work on different parts of the same file (such as different functions in code), easily merge their changes to the file, and resolve any potential conflicts
- Ensuring that there is a central copy of all code, enabling administrators to handle code backups centrally

The most popular code management toolkit is Git, an open-source solution that is freely available. Many vendors, however, have created commercial SCM solutions that use the underlying Git toolset. This means that once you have a good understanding of how Git works, there are

multiple different solutions that you can use with ease that add additional functionality and a user interface to the underlying Git toolset.

Let's look at the AWS service for hosting Git repositories, as well as related services that can be used to create CI/CD pipelines.

AWS CodeCommit

AWS CodeCommit is a cloud-based service that provides a managed source control system to host Git repositories. CodeCommit stores code, binaries, and metadata in a way that provides resilience, redundancy, and high availability. CodeCommit encrypts all files that it stores, and through integration with AWS Identity and Access Management (IAM), you can assign specific permissions to different users for different code repositories.

With CodeCommit you can create a code repository to store code related to the data platform and separate code repositories for each of your data domains (where data producers can store code for their data engineering transforms and pipelines). You can also integrate popular development tools with AWS CodeCommit, such as **AWS Cloud9**, **Visual Studio**, and **Eclipse**. Certain AWS services also offer direct integration with CodeCommit, such as support in **AWS Glue** for managing code for your Glue jobs in CodeCommit (and AWS Glue also supports another popular code management solution, GitHub).

CodeCommit currently offers 5 active users per month at no cost as part of the AWS Free Tier. If you need more than 5 active users, the cost is $1 per additional user per month. This comes with a set limit of storage and Git requests per user per month, and if these limits are exceeded there is an additional charge. See the CodeCommit pricing page at `https://aws.amazon.com/codecommit/pricing/`.

We will use CodeCommit in the hands-on section of this chapter, but first let's have an overview of a related service, AWS CodeBuild.

AWS CodeBuild

AWS CodeBuild is a managed service that can be used to compile source code, run unit tests, and produce software packages ready for deployment.

For example, you can use CodeBuild to deploy an environment where you can use an AWS Glue public Amazon ECR image (Docker container) to run unit tests against your code. Once the tests have run, assuming they complete successfully, you can use CodeBuild to write the updated code to an Amazon S3 bucket.

AWS CodePipeline

AWS CodePipeline is a managed service providing **continuous delivery**, enabling you to automate the steps to deploy and update code and infrastructure for your data platform and products.

For example, you can create a pipeline in CodePipeline that instructs CloudFormation to perform an update on resources, based on an updated CloudFormation template. You can then configure that pipeline to be automatically triggered when a new commit is made to the associated CodeCommit repository and branch combination. By doing this, every time you update the CloudFormation template in CodeCommit, the pipeline will be triggered and will use CloudFormation to deploy the update to the existing resources.

Another example is data producers using CodePipeline to manage updates to their data engineering transform and pipeline code. When a data engineer updates the code for a Glue job and commits that to CodeCommit, this can trigger a pipeline that calls CodeBuild to run unit tests against the Python code, and then writes the updated file to Amazon S3. When Glue jobs run, they read the code file from Amazon S3, so the next time the Glue job is triggered it will use the latest version of the code file placed there by CodeBuild. For an example of this implementation, see the following AWS blog post: `https://aws.amazon.com/blogs/devops/how-to-unit-test-and-deploy-aws-glue-jobs-using-aws-codepipeline/`.

Having reviewed a number of AWS services that help implement a DataOps approach to building a data platform, let's now get hands-on with some of these services.

Hands-on – automated deployment of data platform components and data transformation code

While we do not have space to cover all aspects of building a modern data platform, in this section we will cover how to use various AWS services to deploy some components of a data platform. We start by setting up an AWS CodeCommit repository that will contain all the resources for our data repository (such as Glue ETL scripts and CloudFormation templates). We then use AWS CodePipeline to configure pipeline jobs that push any code or infrastructure changes into our target account.

Setting up a Cloud9 IDE environment

Our first step is to create a **Cloud9** IDE environment, which we can use for writing our code and committing code to a CodeCommit repository. Cloud9 is an AWS service that can be used to provision a managed EC2 instance to provide us with a browser-based **Integrated Development Environment (IDE)** that we can use to write, run, and debug code from within our web browser. Cloud9 environments come preinstalled with various tools that are useful for developing in an AWS environment, such as the **AWS Command Line Interface (CLI)** and a **Git** client.

Let's get started:

1. Log in to the **AWS Management Console** and use the top search bar to search for and open the **Cloud9** service.
2. On the **Cloud9** dashboard page, click on **Create environment** at the top right.
3. For **Name**, provide a name for your environment, such as `dataeng-ide`, and optionally provide a description that explains what you are using the environment for (this will be visible to other AWS users in this account).
4. For **Environment type**, ensure that **New EC2 instance** is selected, and for **Instance type**, select **t2.micro**. If there are no **t2.micro** instances available when you launch your Cloud9 IDE, either wait a while and try again, or use a different instance size (just be aware of the pricing differences for the different instance types).
5. Leave all other settings at their defaults, and click **Create**.
6. On the list of your Cloud9 environments, click on the link to **open** your Cloud9 IDE. Note that it takes a few minutes for your environment to be created.
7. Once the environment has been created, use the terminal at the bottom of the Cloud9 environment to run the following command: `git --version`. This should return the version of Git that is preinstalled in the Cloud9 environment.

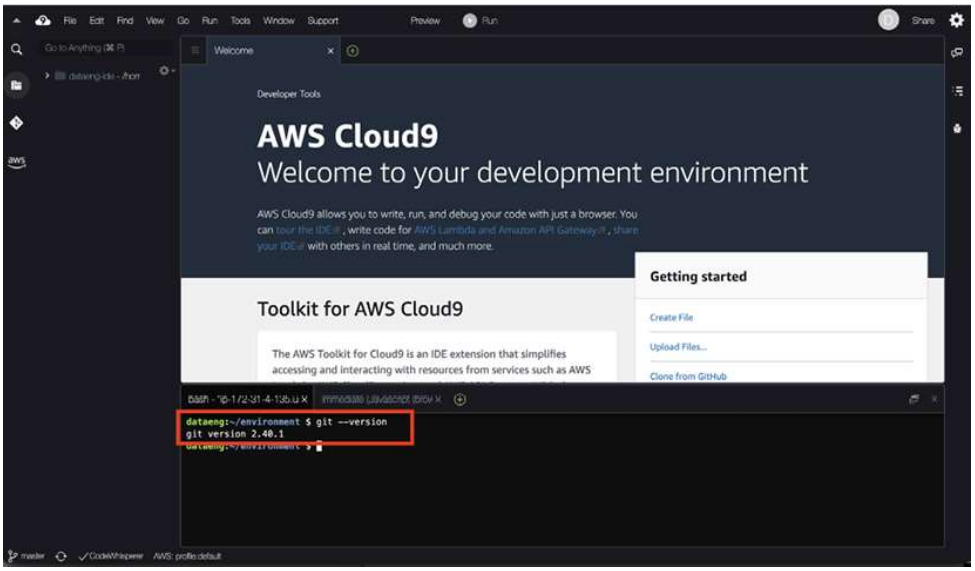

Figure 16.1: AWS Cloud9 console

8. You can now configure a username and email that will be associated with your Git commits by running the following commands in the Cloud9 terminal. Modify the following commands to use a username of your choice and specify your email address:

```
git config --global user.name "Gareth Eagar"
git config --global user.email gareth.eagar@example.com
```

9. We can now configure the AWS CLI credential helper, which will automate the provisioning of credentials needed to authenticate to CodeCommit. Run the following two commands to configure the credential helper:

```
git config --global credential.helper '!aws codecommit credential-helper $@'
git config --global credential.UseHttpPath true
```

10. For the last step in this part of the hands-on exercise, let's create a new directory in our Cloud9 environment to contain our Git repositories. By default, you should be in the `/home/ec2-user/environment` directory in the terminal window, and you can run the following commands to create a new `git` subdirectory, and then change into that directory:

```
mkdir git
cd git
```

Now that we have configured our IDE environment, let's move on to creating our **CodeCommit** repositories.

Setting up our AWS CodeCommit repository

In this step, we create an AWS CodeCommit repository to store a CloudFormation template that deploys a Glue job, and to store PySpark code for the Glue job. We then clone the repository in our Cloud9 environment.

Let's get started:

1. We will be going back to our Cloud9 IDE environment shortly, so use a new browser tab to open the **AWS Management Console** and use the top search bar to search for, and open, the **CodeCommit** service.
2. On the **Repositories** page, click on **Create repository** on the right-hand side.
3. For **Repository name,** set the name to `data-product-film`, and optionally provide a **Description**. Then, click **Create**.
4. In the left-hand side menu, make sure **Repositories** is selected, and then click on the **Clone HTTPS** link under the **Clone URL** column for your `data-product-film` repository. This will copy the HTTPS URL for the repository to our clipboard, which we will need in the next step.
5. Go back to your browser tab with the **Cloud9** IDE environment, and in the terminal/console window, run `pwd` to make sure you are in the `home/ec2-user/environment/git` directory.
6. In the **terminal/console** window, run `git clone <https-repo-link>`, replacing `<https-repo-link>` with the link you copied in *step 5*. For example:

```
git clone https://git-codecommit.us-east-2.amazonaws.com/v1/repos/data-product-film
```

7. In the **terminal/console** window, change into the `data-product-film` directory and create new subfolders – one for our Glue PySpark code, and one for CloudFormation templates:

```
cd data-product-film
mkdir glueETL_code
mkdir cfn_templates
```

8. We also want to create a new S3 bucket to store resources related to our data product. In the **terminal/console** window, run the following command to create a new bucket, but make sure to replace `initials` with your own initials or unique identifier:

```
aws s3 mb s3://data-product-film-initials
```

So far, we have created a repository in AWS CodeCommit, cloned the repository into a Cloud9 IDE environment, and created an Amazon S3 bucket and directories to store our files. We can now move on to our next exercise, where we will add a Glue ETL script and a CloudFormation template into our repository.

Adding a Glue ETL script and CloudFormation template into our repository

In this section, we will create a new CloudFormation template file that defines an AWS Glue job, and will also create a file that contains the code for the Glue job:

1. Log in to the **AWS Management Console** and use the top search bar to search for, and open, the **Cloud9** service.
2. In the list of your environments, click on the `dataeng-ide` environment (which we created earlier), and then click on **Open in Cloud9**. Note that if your Cloud9 environment has been idle for more than 30 minutes, it will have automatically shut down and you will need to wait a few minutes while the environment restarts.
3. Once your Cloud9 environment is ready, click on **File | New File**. Paste the following code into the editor. Note that you can also copy and paste this code from the GitHub site for this chapter of the book:

IMPORTANT: Make sure you change the S3 path to reference the name of the curated zone bucket.

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from awsglue.dynamicframe import DynamicFrame
args = getResolvedOptions(sys.argv, ["JOB_NAME"])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args["JOB_NAME"], args)
# Load the streaming_films table into a Glue DynamicFrame from the Glue catalog
StreamingFilms = glueContext.create_dynamic_frame.from_catalog(
    database="curatedzonedb",
    table_name="streaming_films",
    transformation_ctx="StreamingFilms",
)
# Convert the DynamicFrame to a Spark DataFrame
spark_dataframe = StreamingFilms.toDF()
# Create a SparkSQL table based on the steaming_films table
spark_dataframe.createOrReplaceTempView("streaming_films")
# Create a new DataFrame that records number of streams for each
# category of film
CategoryStreamsDF = glueContext.sql("""
SELECT category_name,
         count(category_name) streams
FROM streaming_films
GROUP BY category_name
""")
# Convert the DataFrame back to a Glue DynamicFrame
CategoryStreamsDyf = DynamicFrame.fromDF(CategoryStreamsDF, glueContext, "CategoryStreamsDyf")
# Prepare to write the dataframe to Amazon S3
############## NOTE ##############
#### Change the path below to
#### reference your bucket name
################################
s3output = glueContext.getSink(
  path="s3://dataeng-curated-zone-gse23/streaming/top_categories",
  connection_type="s3",
  updateBehavior="UPDATE_IN_DATABASE",
  partitionKeys=[],
  compression="snappy",
  enableUpdateCatalog=True,
  transformation_ctx="s3output",
)
# Set the database and table name for where you want this table
# to be registered in the Glue catalog
s3output.setCatalogInfo(
  catalogDatabase="curatedzonedb", catalogTableName="category_streams"
)
# Set the output format to Glue Parquet
s3output.setFormat("glueparquet")
# Write the output to S3 and update the Glue catalog
```

```
s3output.writeFrame(CategoryStreamsDyf)
job.commit()
```

4. Ensure that in the previous step, you modified the name of the S3 bucket on line 47 to match the name of your curated zone bucket, and then click **File | Save As**. Make sure to select the `glueETL_code` directory, give the file a name of `Glue-streaming_views_by_category.py` , and then click **Save:**

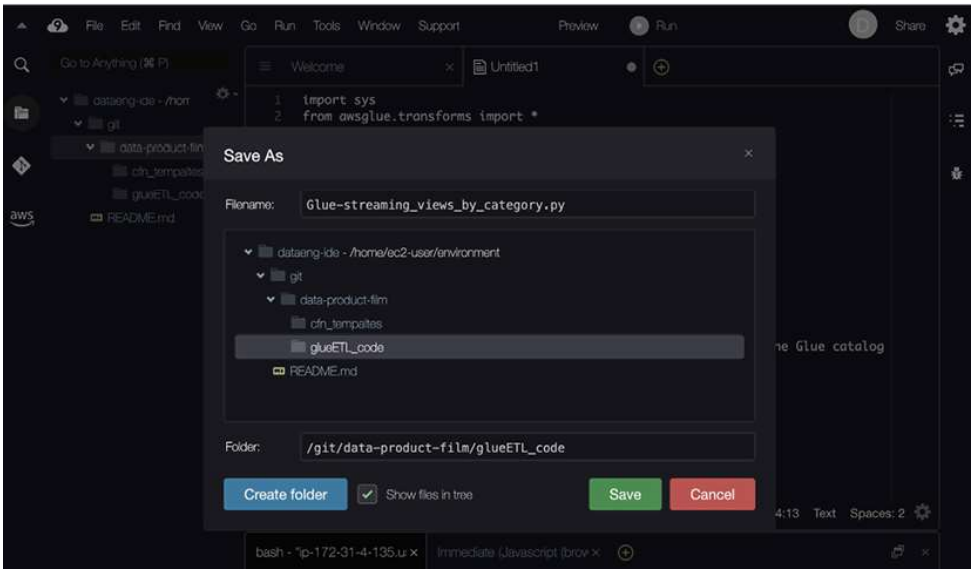

Figure 16.2: Cloud9 – saving the Glue ETL code

5. Now we can create our CloudFormation template, which will be used to deploy a new Glue job. Click on **File | New File**. Paste the following code for our CloudFormation template into the text editor for the new file. Note that you can also copy and paste this code from the GitHub site for this chapter of the book:
IMPORTANT: Make sure you change the S3 script location parameter to reference the name of the bucket that you created earlier in this chapter.

```
AWSTemplateFormatVersion: '2010-09-09'
# CloudFormation template to deploy the streaming view by category
# Glue job.
# In the Parameters section we define parameters that can be passed to
# CloudFormation at deployment time. If no parameters are passed in, then the
# specified default is used.
Parameters:
# JobName: The name of the job to be created
  JobName:
    Type: String
    Default: streaming_views_by_category
# The name of the IAM role that the job assumes. It must have access to data,
# script, and temporary directory. We created this IAM role via the AWS
# console in Chapter 7.
  IAMRoleName:
    Type: String
    Default: DataEngGlueCWS3CuratedZoneRole
# The S3 path where the script for this job is located. Modify the default
# below to reference the specific path for your S3 bucket
  ScriptLocation:
    Type: String
```

```
    Default: "s3://data-product-film-initials/glueETL_code/Glue-streaming_views_by_category.py"
# In the Resources section, we define the AWS resources we want to deploy
# with this CloudFormation template. In our case, it is just a single Glue
# job, but a single template can deploy multiple different AWS resources
Resources:
# Below we define our Glue job, and we substitute parameters in from the
# above section.
  GlueJob:
    Type: AWS::Glue::Job
    Properties:
      Role: !Ref IAMRoleName
      Description: Glue job to calculate number of streams by category
      Command:
        Name: glueetl
        ScriptLocation: !Ref ScriptLocation
      WorkerType: G.1X
      NumberOfWorkers: 2
      GlueVersion: "3.0"
      Name: Streaming Views by Category
```

6. Ensure that in the previous step, you modified the name of the S3 bucket on line 22 to match the name of your data product film bucket, and then click **File | Save As**. Make sure to select the `cfn_templates` directory, give the file a name of `CFN-glue_job-streams_by_category.cfn` , and then click **Save.**
7. Let's now commit the two new files we created to our `data-product-film` repository. To do this, use the **terminal/console** window in Cloud9. Make sure you are in the `/home/ec2-user/environment/git/data-product-film` directory and run the following commands to commit the file to the repository:

```
git add .
git commit -m "Initial commit of CloudFormation template and Glue code for our streaming views by categc
git push
```

> Note that you can type out the above commands or copy and paste them from the GitHub repo for this book, but be careful about copying and pasting from the eBook or PDF versions, as it may use quote characters that will not work correctly when you paste the code into Cloud9.

The above `git add` command stages all files in the current directory and its subdirectories to be added to Git. The `git commit` command prepares the file to be written to our CodeCommit Git repository, along with a message explaining the purpose of this specific commit. The `git push` command takes the prepared files and copies them into the CodeCommit repository.

8. Now, access the **AWS Console** and navigate to the **CodeCommit** service. If you review the `data-platform-film` repository in CodeCommit, you should see the files that we just created in the repository.

We have now created a CloudFormation template and Glue ETL code and committed the files to our CodeCommit repository. Next, we create pipelines to automate the deployment of the Glue job and code.

Automating deployment of our Glue code

When a Glue job runs, it reads the code for the job from Amazon S3. In this section, we will set up a pipeline using AWS CodePipeline that will copy the contents of our data product film repository to Amazon S3 whenever a change is made to any of the files:

1. Log in to the **AWS Management Console** and use the top search bar to search for and open the **CodePipeline** service.
2. On the page listing pipelines, click on **Create pipeline** on the top right-hand side.
3. For **Pipeline name**, enter `data-product-film-resources`, and for **Role name**, accept the default. Leave **Advanced settings** as the default, and then click on **Next**.
4. For **Source provider**, select **AWS CodeCommit** from the drop-down list.
5. For **Repository name**, select **data-product-film**.
6. For **Branch name**, select the default (usually the default branch is called **master**, but it may also be called **main**, depending on the tool that was used to commit the first file to the repository).
7. For **Change detection options**, select **Amazon CloudWatch Events.**
8. Leave other settings as default, and click **Next**.
9. For **Build – optional**, click **Skip build stage**. Then click **Skip** to confirm.
10. For **Deploy provider**, select **Amazon S3** from the drop-down list.
11. For **Bucket**, select the bucket we created earlier for our data product (such as **data-product-film-initials**).
12. Select the **Extract file before deploy** option, leave **Deployment path – optional** blank, then click **Next**.
13. Review the settings, then click **Create pipeline**.
14. Open the **Amazon S3** console and confirm that the files from your repo have now been copied into your `data-product-film-initials` bucket.

Our Glue code is now deployed, so let's now set up a pipeline that will deploy our Glue job using CloudFormation.

Automating the deployment of our Glue job

Let's create a new pipeline that will be triggered whenever the CloudFormation template for our Glue job is updated. In order to do this, we first need to update one of our IAM roles.

When we deploy a CloudFormation template via the console, by default it will use the credentials of the user that we are logged in as to create the required resources (such as a Glue job). However, when doing an automated deployment via CodePipeline, we need to specify an IAM role that has the necessary permissions to deploy the resources instead.

For our use case, we can use the `DataEngGlueCWS3CuratedZoneRole` role, which we created in *Chapter 7* to run our Glue job. However, we need to modify the role so that CloudFormation is able to assume it when deploying resources, which we do by adding a trust relationship for CloudFormation to the role. We also want to add permissions to allow this role (which is also used to run our Glue job) to access the new bucket we created that contains the Glue ETL code:

1. Log in to the **AWS Management Console** and use the top search bar to search for, and open, the **IAM** service.
2. In the left-hand menu, click on **Roles**, and then search for `DataEngGlueCWS3CuratedZoneRole`, and then click on the role name so we can edit it.
3. Select the **Trust relationships** tab, and then click **Edit trust policy.**
4. Modify the **Service** portion of the policy to also include the **CloudFormation** service, as follows:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "cloudformation.amazonaws.com",
                    "glue.amazonaws.com"
                ]
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

5. Once the policy JSON is updated, click on **Update policy**.
6. Click on the **Permissions** tab, expand the `DataEngGlueCWS3CuratedZoneWrite` policy, and click **Edit**.
7. Add your `data-product-film-initials` bucket in the list of resources that provide `s3:GetObject` permissions as follows (making sure to modify the bucket name initials to reflect your bucket name):

```
"Resource": [
    "arn:aws:s3:::dataeng-landing-zone-initials/*",
        "arn:aws:s3:::dataeng-clean-zone-initials/*",
        "arn:aws:s3:::data-product-film-initials/*"
    ]
```

Now that we have permissions configured as needed, let's create the pipeline:

1. Log in to the **AWS Management Console** and use the top search bar to search for, and open, the **CodePipeline** service.
2. On the page listing pipelines, click on **Create pipeline** at the top right-hand side.
3. For **Pipeline name**, enter `data-product-film-glue-streaming-views-by-category-job` , and for **Role name**, accept the default. Leave **Advanced settings** as default, and then click on **Next**.
4. For **Source provider**, select **AWS CodeCommit** from the drop-down list.
5. For **Repository name**, select `data-product-film` .
6. For **Branch name**, select the default (usually the default branch is called **master**, but it may also be called **main**, depending on the tool that was used to commit the first file to the repository).
7. For **Change detection options**, select **Amazon CloudWatch Events.**
8. Leave other settings as default, and click **Next**.
9. For **Build – optional**, click **Skip build stage**. Then click **Skip** to confirm.
10. For **Deploy provider**, select **AWS CloudFormation** from the dropdown.
11. For **Action mode**, select **Create or update a stack.**
12. For **Stack name**, enter `glue-job-streaming-views-by-category-job` .
13. For **Template**, select **SourceArtifact** for **Artifact name**.
14. For **File name**, enter the name of our CloudFormation template: `cfn_templates/CFN-glue_job-streams_by_category.cfn` .
15. For **Role name**, select **DataEngGlueCWS3CuratedZoneRole** from the dropdown.
16. Leave all other settings blank and then click **Next**.
17. Review the settings, then click **Create pipeline**.

18. The pipeline should run, deploying the Glue job to your account. In the **AWS Management Console**, go to the **CloudFormation** service to confirm that the template was deployed, and then go to the **AWS Glue** console to confirm that the job has been created. You can also optionally run the Glue job in order to create the new `top_categories` table.

Testing our CodePipeline

We created our CodePipeline so that when a change to our CloudFormation template (which deploys our Glue job) is committed to CodeCommit, the change will automatically be deployed by CodePipeline.

Let's test it out:

1. Log in to the **AWS Management Console** and use the top search bar to search for, and open, the **CodeCommit** service.
2. Click on the **data-product-film** repository, then click on the **cfn_templates** folder, and then click on the name of our CloudFormation template (`CFN-glue_job-streams_by_category`).
3. Click on the **Edit** button at the top right.
4. Let's increase the number of workers for our Glue job from 2 to 3. Look for the `NumberOfWorkers` attribute (around line 38) and change the number from `2` to `3`.
5. At the bottom of the page, complete the form for **Commit changes to master** by providing details for **Author name**, **Email address**, and **Commit message**.
6. Click on **Commit changes**.
7. Now navigate back to the **CodePipeline** service in the AWS console. When you review the list of pipelines, you should see that your **cloudformation-glue-job-deployment** pipeline is listed as **In progress** (although it may take a few moments before the status updates).
8. Navigate to the CloudFormation service, click on your stack name, then view the **Events** tab and confirm that your Glue job is being updated. You can also optionally go to the AWS Glue console and confirm in the **Job details** tab that the number of workers is now set to **3**.

We've now created a CloudFormation template to deploy a Glue job, and linked that with a CodePipeline pipeline to automatically deploy updates to the job whenever we commit a new version of the CloudFormation template to our CodeCommit repository. With this, we are now managing our code through an SCM system (Git with CodeCommit, in this case), and we can automate the deployment of our pipeline whenever a change is made to our transformation code or the configuration of our AWS Glue Job CloudFormation template.

Summary

In this chapter, we introduced a number of approaches to implementing a modern data platform. We looked at the high-level goals of a data platform (flexibility, scalability, good governance, secure, and self-serve enabled), and then discussed the pros and cons of building versus buying a data platform.

We then discussed how a DataOps approach brings automation to the process of developing data products. We looked at how the deployment and management of infrastructure (such as Glue jobs) can be automated, as well as how code for transforms can be managed with an SCM system.

We then got hands-on with some of the AWS services that can be used to automate data platforms, including CloudFormation, CodeCommit, and CodePipeline. We created a Git repository in CodeCommit and used it to store both the PySpark code for our Glue ETL job and a CloudFormation template to deploy the Glue job. We then looked at how we could create a

pipeline that would automatically deploy our Glue job (as well as any updates to the Glue job configuration). Any updates to our ETL code are also automatically pushed to the Amazon S3 location where the Glue job reads the ETL code, ensuring that updates to the code would be run with any future executions of the Glue job.

A modern data platform should be managed by a central team, and should enable data producers to easily build data products and catalog those in a way that data consumers can find and access the data products. The platform should also make it easy to use modern table formats (such as Apache Iceberg), and data producers should be able to use automation to deploy and manage their transformation code.

There is a lot more to building a modern data platform than we could cover in this chapter, and you are encouraged to read further on this topic. But we'll now move on to the last chapter of this book, where we will wrap up by looking at the bigger picture of analytics, real-world data pipelines, and potential future developments and trends in this space.

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