CSCI 5411
Advance Cloud Architecting

TERM ASSIGNMENT

Banner ID: B00980674
GitHub Link: OvaizAli/AWS-DE-ZoomCamp

Table of Contents

1. Introduction

The goal of this application is to create a data engineering pipeline and analytics solution for NYC's Yellow Cabs. With over 13,500 medallion taxis licensed to operate across the city, these cabs are a vital part of New York's transportation system, providing millions of rides to residents and visitors [1].

Hence, this project uses cloud technologies to build an end-to-end ETL (Extract, Transform and Load) pipeline to achieve the following objectives:

1. Automate data processing by ingesting and transforming large volumes of taxi trip data available online on their website.
2. Clean, transform and organize the data to make it easy to use for analysis and reporting.
3. Provide meaningful insights through interactive dashboards using Looker Studio, helping stakeholders understand trends, operations, and efficiency.
4. Empower data-driven decisions for medallion owners, transportation authorities, dispatch operators, and anyone involved in improving NYC Yellow Taxi services.

2. Application Selection

- Opensource Application: Data Engineering Zoomcamp [2]
- Repository Link: [DataTalksClub/data-engineering-zoomcamp: Free Data Engineering course!]

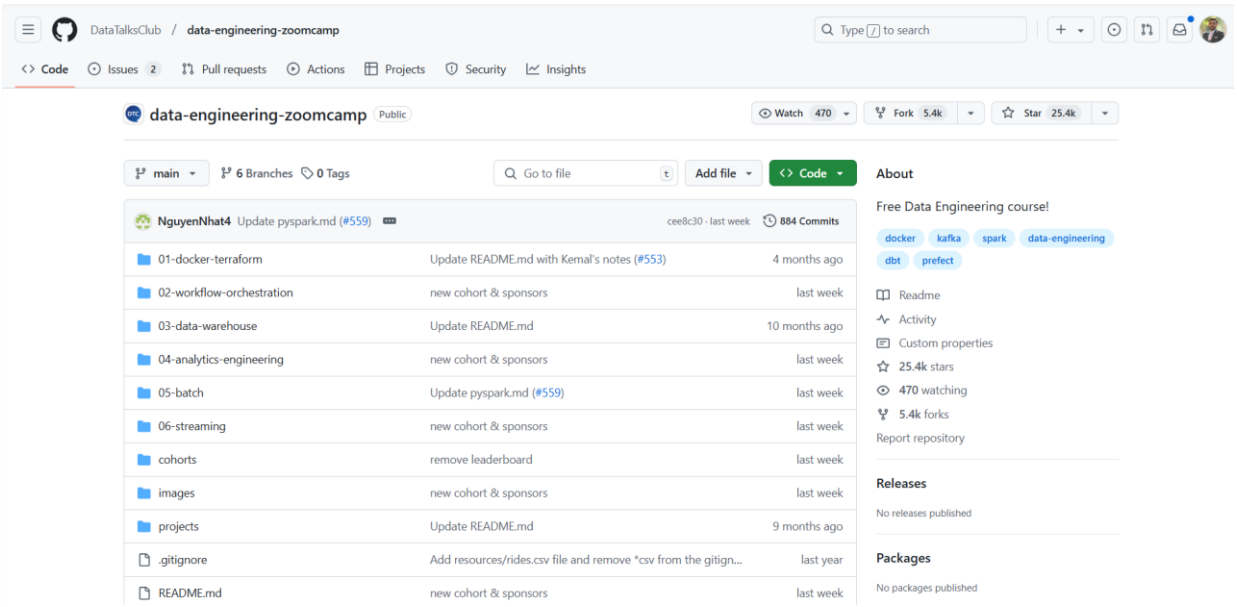

Figure 1: Screenshot of the opensource repository used.

❖ Motivation for Selection

The primary motivation for selecting this open-source application for my term assignment stems from my experience with previous iterations of the Data Engineering Zoomcamp. Those sessions primarily emphasized open-source tools or sponsored technologies focused heavily on Google Cloud Platform (GCP). While reviewing their FAQs, I noticed frequent requests from participants to see the project implemented using AWS services.
Moreover, I identified a gap in their approach: although cloud services were utilized for data engineering, the implementations lacked alignment with the best practices outlined in the AWS Well-Architected Framework. To address this, I chose this repository to

contribute by integrating AWS-based solutions that adhere to these best practices. This effort bridges the gap while supporting my passion for advancing the open-source data analytics community.

❖ **Justification for Selection**

The repository is active and reliable, with **884 commits**, showing it is regularly updated and improved.

Moreover, it is well-known in the data engineering community, with:

- **25.4k stars**, indicating its popularity and support from the open-source community.
- **470 watchers**, showing active interest from users and developers.
- **5.4k forks**, meaning it has been widely adapted and contributed to by others.

Hence, these factual details make the repository an informed choice, meeting the requirements of having at least 100 commits and 50 stars, and it is a great foundation and addition to my portfolio for building data engineering solutions.

3. Selected Cloud Services (AWS and GCP)

❖ AWS Services:

Table 1: Table listing all the AWS services used in the architecture.

Category	Selected Service
Compute	AWS Lambda
Storage	Amazon Simple Storage Service (S3)
Networking and Content Delivery	Amazon VPC
Application Integration	AWS Step Functions
	Amazon EventBridge
	Amazon Simple Notification Service
	Amazon Simple Queue Service
Management and Governance	Amazon CloudWatch
	AWS CloudFormation
Analytics	AWS Glue

❖ GCP Services:

Table 2: Table listing all the GCP services used in the architecture.

Category	Selected Service
Analytics	Big Query
	Looker Studio
Compute	Cloud Function

4. Justification of Chosen Services

- **AWS Services**

1. AWS Lambda

    - AWS Lambda was used to run code in response to events without provisioning or managing servers for prolonged time. Moreover, it scales automatically and charges only for execution time, making it a cost-efficient solution for triggering actions based on data processing events. Since, the requirement of my application was to run the complete ETL once every month to fetch the data from NYC Taxi Api, Lambda was used to fetch and process data, and trigger other services, fitting well with the event-driven architecture of this data pipeline. Hence, using Lambda for this computation was an efficient approach rather than provisioning servers for long time.

2. Amazon Simple Storage Service (S3)

    - As, Amazon S3 provides durable and highly available object storage. S3 was used to store the raw, processed, and final data (in bronze, silver, and gold buckets). Nonetheless, it allows scalable and cost-effective storage while ensuring data durability, as I integrated lifecycle management, bucket versioning, and server-side encryption.

3. Amazon VPC

    - Amazon VPC (Virtual Private Cloud) was used to define a secure network environment for the services mainly compute and storage. It helped me create isolated network segments for services like Lambda, Glue, and Step Functions to communicate securely. This ensures that data flows between services within the same private network while being isolated from the public internet, thus securing sensitive data.

4. AWS Step Functions

    - AWS Step Functions was used to orchestrate the data pipeline by coordinating AWS Lambda functions, Glue jobs, and other monitoring services. It ensures that the pipeline ran smoothly by defining clear workflows and states, managing retries, and providing error handling. This also provided support for both synchronous and asynchronous workflows. Although I initially wanted to implement Amazon Managed Workflows for Apache Airflow (MWAA), due to restrictions on Cloud Academy, I opted for AWS Step Functions instead.

5. Amazon EventBridge

    - EventBridge enabled the event-driven architecture implementation. It was used to trigger the Step Functions-based data pipeline on the 1st of every month. This allows for automatic processing of data without manual intervention, ensuring that the entire pipeline operates efficiently in response to data updates.

6. Amazon Simple Notification Service (SNS)

- SNS was used to send notifications to my email if any failure was encountered in the pipeline. This ensures that stakeholders (such as developers or admins) are promptly alerted for monitoring and troubleshooting, helping maintain operational efficiency.

7. Amazon Simple Queue Service (SQS)

    - SQS, being a message queuing service, was integrated with Step Functions to act as a Dead Letter Queue (DLQ) for handling failed tasks. This provides fault tolerance to the pipeline and ensures that failed processes can be examined later for troubleshooting and resolution.

8. Amazon CloudWatch

    CloudWatch was used to monitor and log the performance of the data pipeline. It tracks metrics such as Lambda execution times, Glue job statuses, and Step Function executions. CloudWatch Alarms and Logs help detect and diagnose issues. In case of a failure, notifications are sent via SNS to email. This ensures that stakeholders are promptly alerted, enabling them to take the necessary actions to maintain the system's health and efficiency.

9. AWS CloudFormation

    - CloudFormation enabled Infrastructure as Code (IaC) by allowing me to define and provision all AWS infrastructure components in a template. Thus, it ensures repeatable, automated deployments and simplified managing and updating the AWS environment without any manual intervention.

10. AWS Glue

    - AWS Glue was used to perform ETL (Extract, Transform, Load) operations on the data. It helped clean, structure, and transform the raw data stored in S3 (Bronze) into a format suitable for analysis (Silver and Gold). Glue seamlessly integrated with other AWS services like Lambda, S3, and Redshift for efficient data processing. Additionally, Glue's support for PySpark allowed for distributed processing. By enabling Glue to run on spot instances, I was able to reduce costs while maintaining scalability and performance.

❖ **GCP Services**

1. BigQuery

    - Due to issues with Amazon Redshift on Cloud Academy and its associated cost, I opted for Google BigQuery instead. BigQuery efficiently stores and analyzes large volumes of taxi trip data. Moreover, it provides real-time analytics and automatically scales with increasing data volume. Additionally, its integration with Looker Studio enables powerful querying and visualization, empowering decision-makers and stakeholders with actionable insights.

2. Looker Studio
    - Looker Studio was used as an alternative to Amazon QuickSight, as QuickSight was not available in the Cloud Academy environment. Its user-

friendly dashboards and reports enable stakeholders to gain meaningful insights into taxi operations, trends, and efficiency, facilitating data-driven decision-making.

3. Google Cloud Functions (GCF)
    - GCF was used to automate the process of loading data from the S3 Gold bucket into BigQuery. This serverless service integrates AWS and GCP without manual intervention and supports scalable and event-driven operations, aligning with the architecture's automation requirements.

The chosen AWS and GCP services effectively meet the requirements and scope of my data analytics application by ensuring:

- Scalable and automated data processing through Lambda, Google Cloud Function, S3, Glue, and BigQuery.
- Data transformation and structuring to clean, organize, and prepare data for analysis in Looker Studio.
- Data visualization and insights using BigQuery and Looker Studio, empowering stakeholders to track trends and make decisions based on the dashboard.
- Fault-tolerant architecture with services like Step Functions, SQS, and SNS for error handling and notifications.
- Monitoring and performance tracking of the data pipeline through CloudWatch, ensuring the pipeline runs efficiently and issues are promptly detected and resolved.

By incorporating these services, the solution is cost-effective, scalable, and optimized for processing large datasets, ensuring high availability, fault tolerance, and ease of monitoring.

5. Implementation of the AWS Well-Architected Framework principles

Throughout the implementation of this architecture, I adhered to the AWS Well-Architected Framework, focusing on its six key pillars: Operational Excellence, Security, Reliability, Performance Efficiency, Cost Optimization, and Sustainability. Here is how the framework principles were applied to the architecture and implementation of the NYC Taxi Data Pipeline:

1. Operational Excellence

- Incorporating Automation and Orchestration:
    - AWS Step Functions were implemented to orchestrate the ETL pipeline by coordinating services such as Lambda, Glue, and S3.
    - AWS EventBridge rule was scheduled to automate the pipeline's execution on the 1st of every month, minimizing manual intervention and ensuring consistency.

- Monitoring and Alerts:
    - CloudWatch was implemented to provide metrics and logs for Lambda execution times, Glue job statuses, and Step Functions.
    - Nonetheless, alarms send notifications via SNS (email) for failures, ensuring swift response and troubleshooting.

- Infrastructure as Code (IaC):
    - CloudFormation templates was implemented to automate provisioning and management of resources, ensuring repeatable and consistent deployments.

2. Security

- Data Protection:
  - Data in S3 was secured with server-side encryption, ensuring confidentiality with the stored data.
  - Bucket versioning and lifecycle management policies were implemented to enhance durability and compliance.

- Network Isolation:
  - Amazon VPC was incorporated to ensure all communication between services like Lambda, Glue, and Step Functions occurs within a private, secure network, preventing unauthorized access.

3. Reliability

- Fault Tolerance:
  - SQS was included in the architecture as a Dead Letter Queue (DLQ) for Step Functions, capturing all failed tasks for later analysis. (It's generally part of ETL system's)

- Retry Mechanisms:
  - Step Functions incorporate retry strategies for temporary failures in Lambda or Glue jobs, ensuring the ETL pipeline is successfully completed.

- Event-Driven Architecture:
  - EventBridge rule automates the pipeline execution which reduces the risk of missed schedules or manual errors as the manual interaction has been replaced.

- Monitoring and Backup:
  - CloudWatch Logs track execution details, supporting in failure detection and recovery. Nonetheless, the log groups are created for each of the resources to track their individual logs and even subscribing SNS to send an email on the pipeline failure.

4. Performance Efficiency

- Scalable Compute:
  - With respect to compute, AWS Lambda was used as it scales automatically to handle data fetching and processing without the need for provisioning servers.

- Optimized Data Storage:
  - S3's tiered storage architecture (bronze, silver, gold buckets) was implemented as it organizes raw, transformed, and final data for efficient access.

- Distributed ETL Processing:
  - AWS Glue was used as it can leverage PySpark for distributed data transformation, ensuring high performance even with large datasets.

5. Cost Optimization

- Pay-As-You-Go Services:

    - According to my use case where the pipeline runs monthly, I tried to use services which are pay as you go, like Lambda charges based on execution time, which eliminates the need for always-on servers.
    - Moreover, glue jobs running on spot instances, significantly reduces the ETL costs while maintaining reliability.

- Lifecycle Policies in S3:
    - Implemented to transition data to cheaper storage classes over time (for the data not frequently accessed), reducing storage costs.

- Resource Monitoring:
    - CloudWatch alarms tracks resource usage, which would help to identify inefficiencies and optimize spending based on the resources consumed by individual services in the architecture.

6. Sustainability

- Efficient Resource Utilization:
    - Incorporating serverless services like Lambda and BigQuery minimize infrastructure waste by scaling them dynamically based on the workload.

- Spot Instance Usage:
    - Glue jobs configured to utilize spot instances for ETL would reduce the overall carbon footprint and cost.

- Data Organization:
    - Storing and processing data in structured formats with a end to end data pipeline, it would reduce the computational overhead, enhancing energy efficiency.

Hence, the architecture I developed incorporates AWS Well-Architected Framework principles as much possible in the AWS cloud academy lab environment. By using serverless computing, tiered storage, secure network design, and automated monitoring and orchestration, the solution is cost-efficient, secure, scalable, and sustainable. The integration of SNS alerts and SQS DLQs ensures reliability, while CloudFormation ensured operational consistency and agility in the architecture. Therefore, I believe these practices make the pipeline robust, maintainable, and well-aligned with AWS best practices.

6. Best Practices

Throughout the development of the end-to-end project, I incorporated best practices, and the following are the ones implemented in my architecture:

1. Isolated the resources within a dedicated VPC, enabling DNS support and hostnames for better network management.
2. Defined restrictive security groups to ensure controlled access, allowing only necessary inbound and outbound traffic.
3. Separated public and private subnets to isolate sensitive resources from public internet access.
4. Set up an internet gateway for public access and a NAT gateway for secure outbound access from private subnets.
5. Enabled encryption, versioning, and lifecycle policies to manage data retention and

security, while blocking public access for sensitive data buckets.

6. Incorporated AWS Secrets Manager for secure storage of sensitive information, such as queue URLs, with VPC endpoints for private access.
7. Configured Lambda functions with VPC access for secure execution in private subnets, minimizing exposure to the public internet.
8. Incorporated robust error handling with retries and dead-letter queues (DLQs) for better fault tolerance and logging.
9. Configured Glue jobs for ETL processing, optimized with continuous logging, job bookmarks, and spot instance usage for cost efficiency.
10. Step Functions for orchestrating Lambda and Glue workflows, ensuring reliable execution with retry logic and error handling.
11. Set up CloudWatch alarms for monitoring critical workflows, with notifications triggered on failures or anomalies.
12. Automated monthly execution of workflows using Amazon EventBridge with a cron expression, ensuring timely data processing.
13. Enabled spot instances for Glue jobs and optimized Lambda memory and timeout settings to control costs.
14. Enforced strict access controls on S3 buckets and utilized server-side encryption (SSE) for all stored data.

Therefore, I am confident that the approach I have incorporated would ensures a secure, reliable, and cost-efficient data processing pipeline in AWS.

7. Architecture Diagram

✓ Drive link of Architecture Diagram

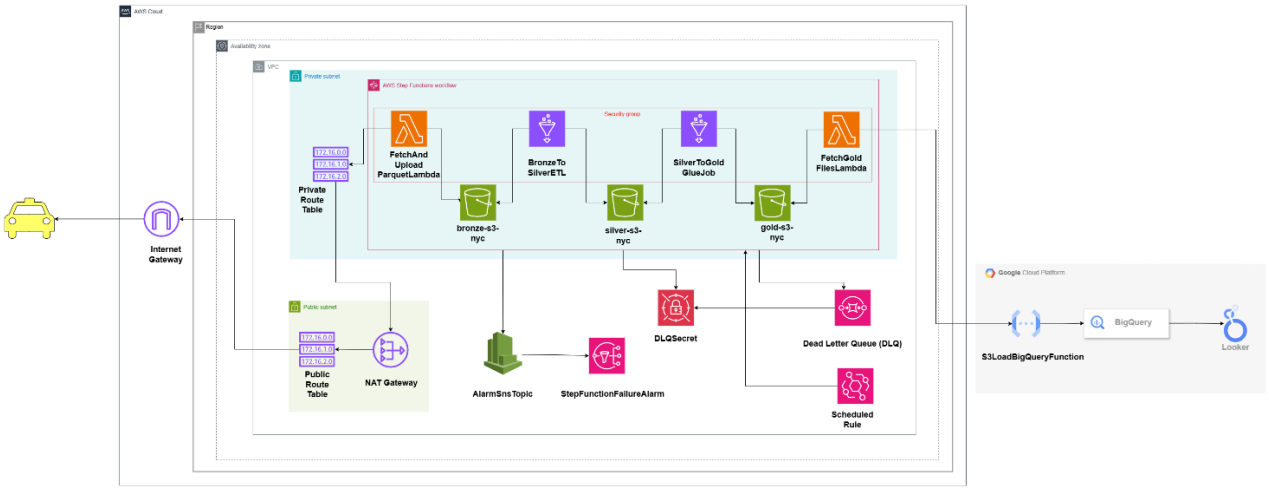

Figure 2: Architecture Diagram.

8. Data Interaction Sequence Diagram

✓ Drive link of Data Interaction Sequence Diagram

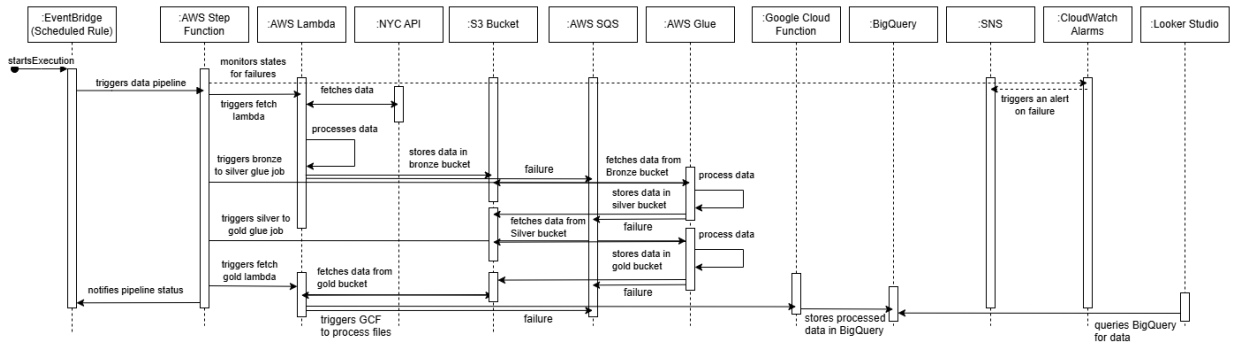

Figure 3: Data Interaction Sequence Diagram

9. Deployed End User Dashboard

Public Dashboard URL: https://lookerstudio.google.com/reporting/536592fb-0fb0-4e24-9183-458968ae1678

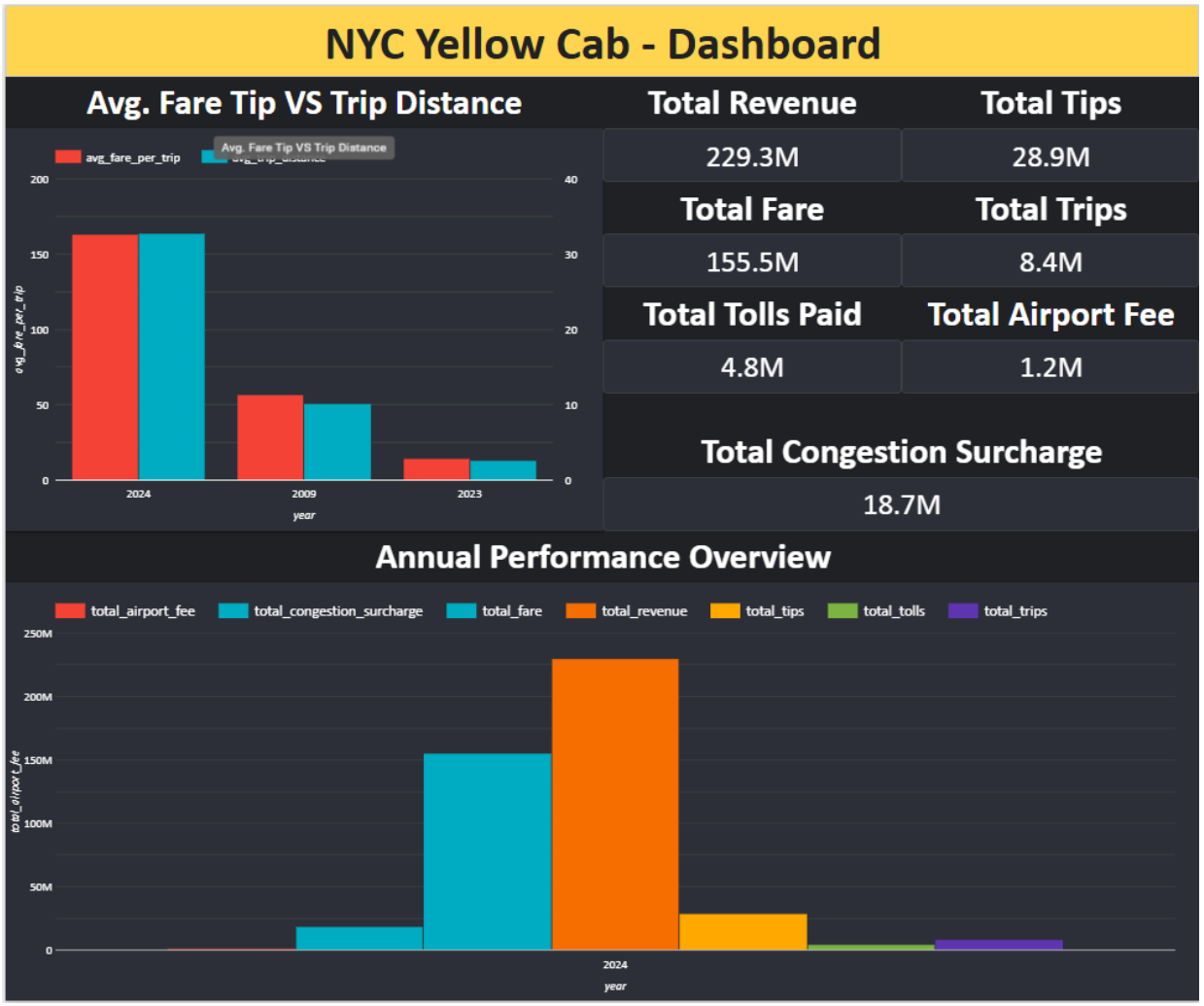

Figure 4: Deployed end user dashboard using Looker Studio.

10. Estimated Cost Analysis

Designing good architecture is not possible without analyzing the running costs of that architecture on the cloud. Hence, here's a breakdown of the estimated costs for the AWS services used in the architecture:

1. **AWS Lambda**:
    - AWS Lambda is priced based on the number of requests and the duration of execution.
        - Requests: $0.20 per 1 million requests.
        - Duration: $0.00001667 per GB-second (the time the function runs, multiplied by the amount of memory allocated).

    - For a function that processes 10,000 records per day, running for 100ms with 128MB of memory, the estimated cost would be:
        - Requests: 10,000 * 30 days = 300,000 requests → $0.06
        - Duration: 100ms * 128MB * 300,000 requests = 3,840,000MB-seconds → $0.064
        - Total Lambda Cost: $0.06 + $0.064 = **$0.124/month**

2. **AWS Glue**:
    - Glue charges for the number of Data Processing Units (DPU) used, and the duration of the ETL job.
        - $0.44 per DPU-hour.

    - For a 1 DPU Glue job running for 1 hour daily:
        - DPU Hours: 1 * 30 days = 30 DPU-hours → $13.20
        - Total Glue Cost: **$13.20/month**

3. **Amazon S3**:
    - S3 costs depend on storage, data transfer, and PUT requests.
        - S3 Standard storage: $0.023 per GB.
        - PUT requests: $0.005 per 1,000 requests.

    - For storing 100GB of data and 10,000 PUT requests per day:
        - Storage: 100GB * $0.023 = $2.30/month.
        - PUT Requests: 10,000 * 30 days = 300,000 requests → $1.50/month.
        - Total S3 Cost: $2.30 + $1.50 = **$3.80/month**

4. **Amazon VPC**:
    - The costs for VPC mainly come from data transfer and use of resources like NAT Gateway.
        - NAT Gateway: $0.045 per hour, plus data processing charges.

    - 1 NAT Gateway running 24/7:
        - NAT Gateway Hours: 24 * 30 = 720 hours → $32.40.
        - Total VPC Cost: **$32.40/month**

5. **Amazon CloudWatch**:
    - CloudWatch charges based on metrics, logs, and alarms.
        - Basic monitoring: Free for most AWS resources.

        - Custom metrics and logs: $0.30 per custom metric per month.
        - Alarms: $0.10 per alarm per month.

    - For 10 custom metrics and 5 CloudWatch Alarms:
        - Custom Metrics: 10 * $0.30 = $3.00.
        - Alarms: 5 * $0.10 = $0.50.
        - Total CloudWatch Cost: **$3.50/month**

6. **Amazon SNS and SQS**:
    - SNS and SQS are priced based on the number of requests and data transferred.
        - SNS: $0.50 per 1 million requests.
        - SQS: $0.40 per million requests.

    - For 1 million SNS messages and 1 million SQS requests per month:
        - SNS: $0.50.
        - SQS: $0.40.
        - Total SNS and SQS Cost: **$0.90/month**

7. **Step Functions**:
    - Step Functions are priced based on the number of state transitions.
        - $0.025 per 1,000 state transitions.

    - For 100,000 state transitions per month:
        - State Transitions: 100,000 * $0.025 = **$2.50/month**

❖ Total Estimated Monthly Cost [4]:

Service	Estimated Monthly Cost
AWS Lambda	$0.12
AWS Glue	$13.20
Amazon S3	$3.80
Amazon VPC	$32.40
Amazon CloudWatch	$3.50
SNS & SQS	$0.90
AWS Step Functions	$2.50
Total	$56.42/month

11. Infrastructure as Code (IaC)

To implement Infrastructure as Code (IaC), I chose AWS CloudFormation, because it is native service provided by AWS. Moreover, CloudFormation is tightly integrated with the AWS ecosystem, which enables the definition and management of AWS resources using declarative templates. It was chosen for the following reasons:

1. CloudFormation works effortlessly with AWS services, which ensures smooth and reliable infrastructure management and deployment.

2. Resources are defined using YAML or JSON templates, which simplifies the infrastructure setup and provides a clear, readable format.

3. CloudFormation also enables tracking changes through stacks, making updates straightforward and providing rollback capabilities for error recovery during the

architecture creation.

4. The service itself is free to use, with costs incurred only for the resources provisioned so it's better to make use of its capabilities.

The CloudFormation script was used to provision an end-to-end architecture that includes networking, storage, compute, data processing, and orchestration resources mentioned in the above sections. The implementation of my cloud formation script involves the following key steps to provision the resources in my application architecture:

1. **Networking**:
    - A VPC with public and private subnets.
    - An Internet Gateway for external connectivity.
    - A NAT Gateway for secure outbound internet access from private subnets.
    - Security Groups to define inbound and outbound traffic rules.
    - VPC endpoints for secure communication with S3 and Secrets Manager.
2. **Storage**:
    - Three S3 buckets (bronze, silver, and gold) for staging data at different processing levels.
    - Lifecycle policies to transition and archive data for cost optimization.
    - Encryption for data security.
3. **Data Processing**:
    - AWS Glue jobs to transform data from bronze to silver and silver to gold stages.
    - ETL scripts are referenced from pre-existing S3 locations.
4. **Compute**:
    - Two Lambda functions:
        - FetchAndUploadParquetLambda fetches data and stores it in the Bronze bucket.
        - FetchGoldFilesLambda processes gold-stage files and triggers further workflows.
5. **Orchestration**:
    - AWS Step Functions orchestrates the entire pipeline, handling retries, waits, and error management.
    - An SQS Dead Letter Queue (DLQ) captures failures for debugging and alerting.
6. **Scheduling**:

- An EventBridge rule triggers the Step Functions workflow on the 1st day of every month.

The important resources from my cloud formation script are listed below in detail for your reference:

Networking (VPC) Configuration

- Created with the CIDR block 10.0.0.0/16 and DNS support enabled.
- Subnets:
  - Public Subnet (10.0.1.0/24), connected to the Internet Gateway.
  - Private Subnet (10.0.2.0/24), connected to the NAT Gateway for internet access.
- NAT Gateway placed in the public subnet with an Elastic IP for internet access by private subnet resources.
- Route Tables configured for public and private subnets, routing internet traffic through the Internet Gateway and NAT Gateway.
- VPC Endpoints: S3 VPC Endpoint to provide private access to S3 from the private subnet.

Security

- Security Groups to define rules for Lambda functions to access resources within the VPC, like S3, Glue, and Redshift.
- Secrets Manager to store sensitive data (e.g., DLQ URL) and is accessed securely by Lambda functions with a **VPC Endpoint** for secure retrieval.

Storage (S3 Buckets)

- Bronze Bucket to store raw data with versioning, encryption, and lifecycle rules (data transitioned to Intelligent-Tiering after 30 days, Glacier after 365 days, and expired after 730 days).
- Silver Bucket to store processed data, with similar settings as the Bronze bucket.
- Gold Bucket to store refined data with access policies ensuring restricted public access.

Lambda Functions

- FetchAndUploadParquetLambda fetches external data, converts it into Parquet format, and uploads it to the Bronze S3 Bucket. Configured with a Dead Letter Queue (DLQ) for failure handling.
- FetchGoldFilesLambda fetches data from the Gold S3 Bucket and invokes a Google Cloud Function (GCF) to load data into BigQuery.

Glue Jobs

- BronzeToSilverGlueJob processes data from the Bronze S3 Bucket to the Silver S3 Bucket, using Spot Instances for cost efficiency.
- SilverToGoldGlueJob processes data from the Silver S3 Bucket to the Gold S3 Bucket.

Step Functions

- DataPipeline would Orchestrate the execution of the Lambda functions and Glue Jobs in a sequence:
    1. Invoke FetchAndUploadParquetLambda.
    2. Wait for 2 minutes before running the BronzeToSilverGlueJob.
    3. Wait for 5 minutes before running the SilverToGoldGlueJob.
    4. Wait for 1 minute before invoking FetchGoldFilesLambda.
    5. A failure handler invokes a Lambda function for any errors and sends logs to the DLQ.

Scheduled Events

- ScheduledRule, a cron job triggers the DataPipeline on the 1st day of every month at midnight UTC.
- CloudWatch Alarm monitors the Step Functions state machine for failures and triggers notifications.

To deploy my AWS CloudFormation template, I used the AWS Management Console and followed the following steps [3]:

1. **Create a New Stack:**
    - Clicked on "Create stack" and selected "With new resources (standard)" to deploy new infrastructure.
2. **Upload the Template:**
    - Uploaded the CloudFormation template in YAML format from the local system. Alternatively, templates can be referenced via an Amazon S3 URL if stored in S3.
3. **Specify Stack Details:**
    - Provided a stack name to identify the deployment.
    - Entered any required parameters defined in the template, such as resource names, instance types, or configuration details.
4. **Configure Stack Options:**
    - Defined advanced options, such as adding tags for resource identification, enabling stack policies, or setting IAM role permissions for CloudFormation.
5. **Review and Deploy:**

- Reviewed the stack configuration to ensure the parameters and resources matched requirements.
- Acknowledged the required IAM role permissions by checking the capabilities acknowledgment box (if the template creates IAM resources).
- Clicked "Create stack" to start the deployment.

6. **Monitor the Deployment:**
   - Monitored the progress of the deployment in the Events tab of the CloudFormation dashboard.
   - Verified the successful creation of resources in the Resources tab.
7. **Validate Deployment:**
   - Cross-checked the provisioned resources in the respective AWS services (e.g., S3, Lambda, VPC) to confirm they were created and configured correctly.

Thus, this approach was used to ensure that the CloudFormation stack deployment was uniform and replicated whenever needed.

12. Challenges Faced

It would be notable to mention some of the challenges I faced during the development as they were important for me to learn and move on to other alternatives to overcome them.

1. Encountered challenges while integration between AWS Redshift and other services as the cloud academy had some IAM role issue. Thus I opted for Big Query as my data warehouse alternative.

2. Faced limitations from the cloud academy side that I was not able to use with QuickSight as it was not offered under ther learner lab edition, hence I switched to Looker Studio.

3. Encountered challenges in tracking and managing the costs of AWS services, especially with unpredictable usage spikes during testing phases. Moreover, while testing Redshift I lost 50% of my credits and end up knowing it was not accessible in cloud academy.

13. Future Work

Moreover, upon analyzing my current architecture for future work, I will prioritize the following actions to ensure the architecture is robust:

1. **Security:**
   - Implement least-privilege IAM roles to minimize access rights.
   - Utilize VPC Endpoints for private, secure service access within the VPC.
2. **Infrastructure as Code (IaC):**
   - Break down CloudFormation templates into smaller, modular, and reusable stacks.

- Automate IaC testing using tools like TaskCat to validate templates before deployment.

In addition to these priority improvements, I remain open to further consultation and will continue working on refining the architecture over time.

14. References

[1] NYC.gov, "Yellow Cab Information," NYC, 2024. [Online]. Available: https://www.nyc.gov/site/tlc/businesses/yellow-cab.page. [Accessed: Dec. 4, 2024].

[2] GitHub, "DataTalksClub: Data Engineering Zoomcamp," GitHub, 2024. [Online]. Available: https://github.com/DataTalksClub/data-engineering-zoomcamp. [Accessed: Dec. 4, 2024].

[3] Amazon Web Services, "AWS CloudFormation User Guide," AWS, 2024. [Online]. Available: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html. [Accessed: Dec. 4, 2024].

[4] Amazon Web Services, "AWS Pricing Calculator," [Online]. Available: https://docs.aws.amazon.com/pricing-calculator/. [Accessed: Dec. 4, 2024].