A Project Report on

Employee Data Management System

PREPARED BY:

Associate ID	Associate Name
2388066	Tanish Waghule
2387739	Aryan Khokle
2387763	Gummidi Pudi Lalithya
2388081	Sarthak Sonar
2388108	Proshanjeet Chanda
2388111	Aditya Tighare
2388080	Yatiraj Kulat

GUIDED BY:

Mentor

Kavin Kumar G
Associate
AIA – Cloud Data Integration

Coach

Yalin Maria
Contractor
CPO L&D GenC Training

CONTENTS

LIST OF ABBREVIATIONS

AWS	Amazon Web Services
INT	Integer
DB	Database
S3	Simple Storage Service
CSV	Comma- Separated Values
ID	Identity
AZ	Availability Zones
IAM	Identity And access management
ETL	Extract, Transform and Load
SQL	Structured Query Language
API	Application Programming Interface

LIST OF FIGURES

LIST OF TABLES

Introduction

The Employee Data Management System project is designed to create a comprehensive and efficient solution for managing employee records within an organization. This system leverages a suite of AWS services to handle the entire lifecycle of employee data, from extraction and transformation to loading, storage, and archival. By automating these processes, the system aims to enhance data accuracy, accessibility, and overall management efficiency. The project addresses the need for a scalable and reliable system that can process large volumes of employee data submitted daily by various departments. This data, typically provided in CSV format, includes critical information such as employee IDs, names, departments, join dates, salaries, and email addresses. The system's architecture ensures that this data is processed in a streamlined manner, facilitating quick retrieval for operational needs and comprehensive analysis for strategic decision-making.

Objective:

Develop a simple Employee Data Management System that processes and stores employee records using AWS services (EC2, Lambda, Redshift, DynamoDB, Teradata). The system will handle data extraction, transformation, and loading (ETL) to enable quick retrieval, analysis, and archival of employee information.

Business Case Overview:

A company wants to manage employee data across multiple departments. Each department submits daily employee data in CSV format, containing new employee information. The company needs a system that can:

1. Extract the daily employee data.
2. Transform the data (validate, clean, and enrich it).
3. Load the data into an AWS Redshift warehouse for reporting and analytics.
4. Store the raw data in DynamoDB for quick lookups.
5. Archive the data in Teradata for long-term storage and backup.

Source Requirements:

Source Data (CSV Files):

The data is provided daily in CSV files containing the following fields:

Employee_ID	Name	Department	Join_Date	Salary	Email
1	Alice	HR	2025-01-01	55000	alice@example.com
2	Bob	IT	2025-01-02	60000	bob@example.com
3	Charlie	Finance	2025-01-02	65000	charlie@example.com
4	David	IT	2025-01-03	70000	david@example.com

Target Requirements:

1. AWS S3:

- Store the raw CSV files temporarily in S3 for processing.
- S3 Path: /raw-data/employee/

2. AWS Redshift:

- Store transformed and aggregated employee data for reporting and analytics.

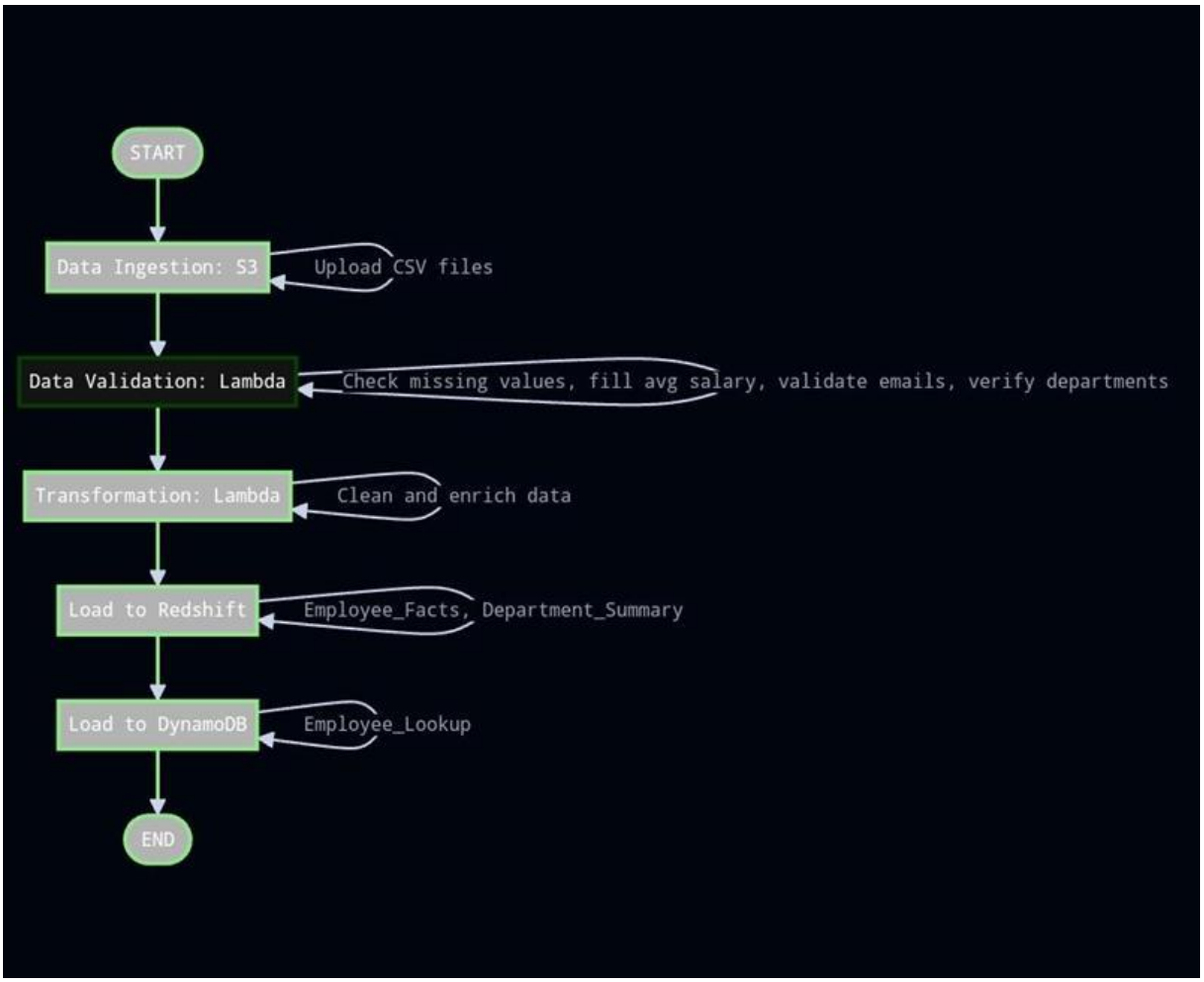
Tables:

- Employee_Facts: Store employee data (ID, name, department, salary, etc.).
- Department_Summary: Aggregated data (total employees and average salary by department).

3. DynamoDB:

- Store employee data for quick lookups by employee ID.
- DynamoDB Table: Employee_Lookup
- Primary Key: Employee_ID

FLOW CHART:



Functional Requirement: 01

- **Creating the S3 Bucket.**

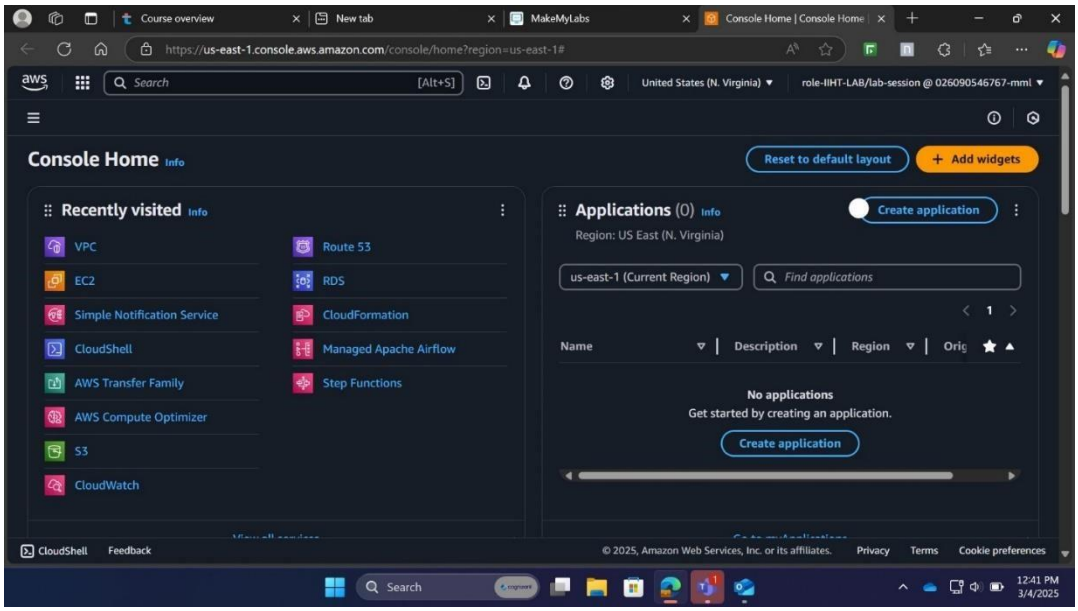


Fig 1: AWS Home page

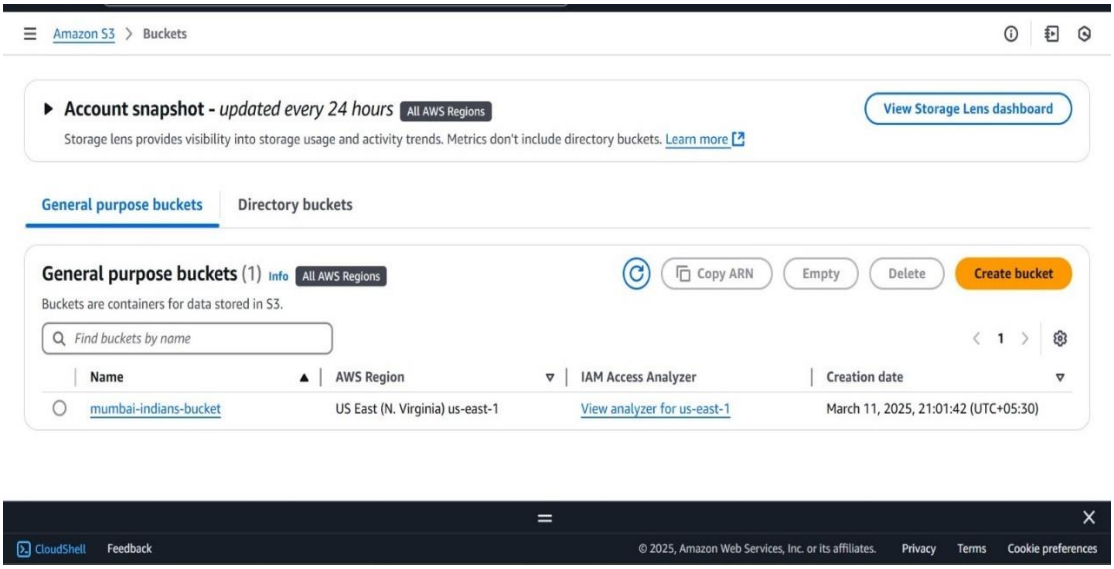


Fig 2: Create S3 bucket

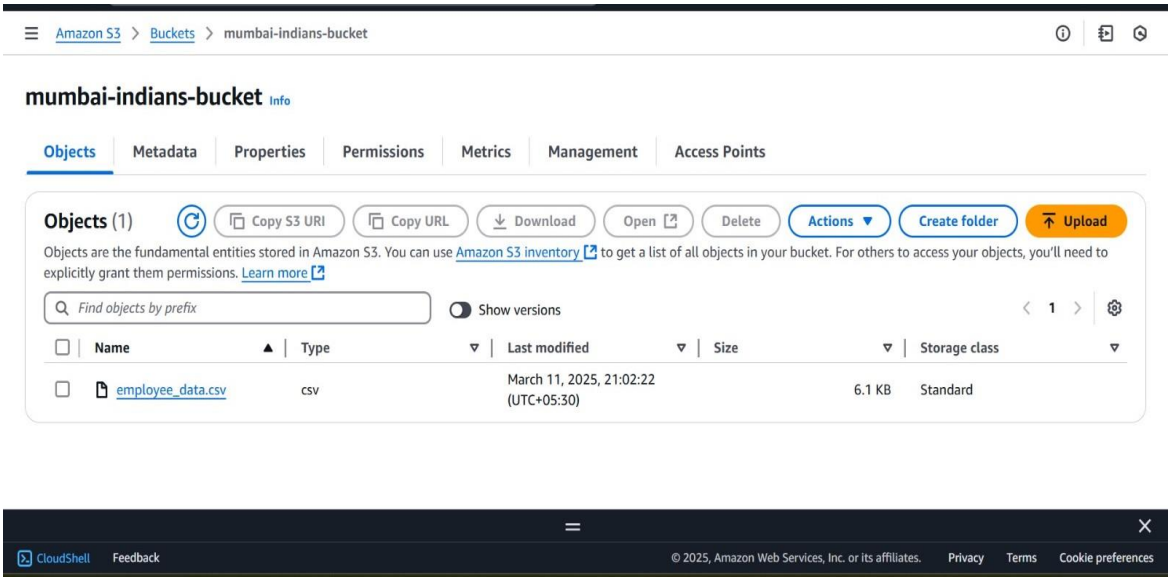


Fig 3: Uploading object into bucket

Creation of "Mumbai-Indians-Bucket" Bucket in Amazon S3

To organize and manage the data efficiently, a new bucket named **"Mumbai-Indians-Bucket"** was created in Amazon S3. This bucket serves as a dedicated storage space for all files related to the Mumbai Indians project. The creation process involved the following steps:

1. **Accessing Amazon S3**: Logged into the AWS Management Console and navigated to the S3 service.
2. **Creating the Bucket**:

- Clicked on the "Create bucket" button.
- Entered "Mumbai-Indians-bucket" as the bucket name.
- Selected the appropriate region for optimal performance and compliance.
- Configured bucket settings, including versioning, encryption, and access permissions.
- Completed the bucket creation process.

Uploading Files to "Mumbai-Indians-bucket"

Once the bucket was created, various files were uploaded to it to ensure all project-related data is stored securely and can be accessed easily. The upload process included:

1. **Selecting Files**: Identified and selected the files to be uploaded, ensuring they are relevant to the Mumbai Indians project.

2. **Uploading Process**:

- Navigated to the "Mumbai-Indians-bucket".
- Clicked on the "Upload" button.
- Dragged and dropped the selected files or used the file picker to choose files from the local system.
- Configured upload settings, such as storage class and encryption.
- Initiated the upload and verified the successful transfer of files.

Functional Requirement: 02

- **creating the IAM role.**

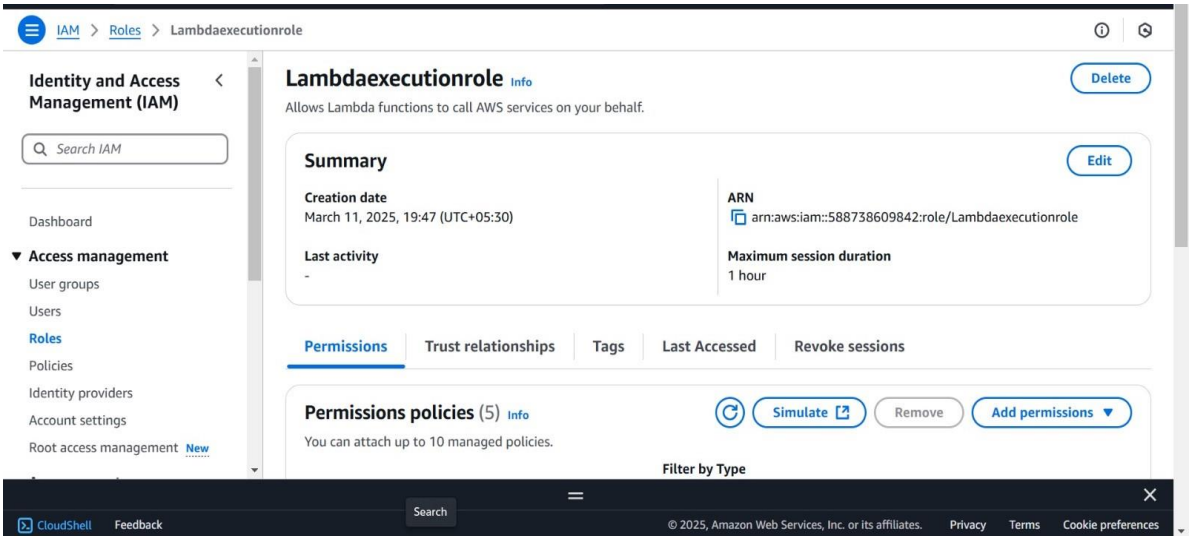


Fig 4: creating an IAM role

Creating an IAM Role for Lambda with Specific Permissions

To enable a Lambda function to interact with various AWS services, an IAM role with the necessary permissions must be created. Here's a detailed description of the process:

1. **Accessing IAM Service**:

- Logged into the AWS Management Console.
- Navigated to the IAM (Identity and Access Management) service.

2. **Creating the IAM Role**:

- Clicked on "Roles" in the IAM dashboard.
- Selected "Create role".
- Chose "AWS service" as the trusted entity and selected "Lambda" as the use case.

3. **Attaching Policies**:

- Attached the following managed policies to grant the required permissions:
- **AmazonRedshiftFullAccess**: Allows full access to Amazon Redshift.
- **AmazonS3FullAccess**: Allows full access to Amazon S3.
- **AmazonDynamoDBFullAccess**: Allows full access to Amazon DynamoDB.
- **SecretsManagerReadWrite**: Allows read and write access to AWS Secrets Manager.
- **AWSLambdaBasicExecutionRole**: Provides basic Lambda execution permissions, including logging to CloudWatch.

4. **Review and Create**:

- Reviewed the role details to ensure all necessary policies were attached.
- Provided a name for the role, such as "LambdaFullAccessRole".
- Clicked "Create role" to finalize the process.

Functional Requirement: 03

- **Generating the Lambda Function for Enriching the Data and then delivering data to AWS redshift and AWS DynamoDB.**

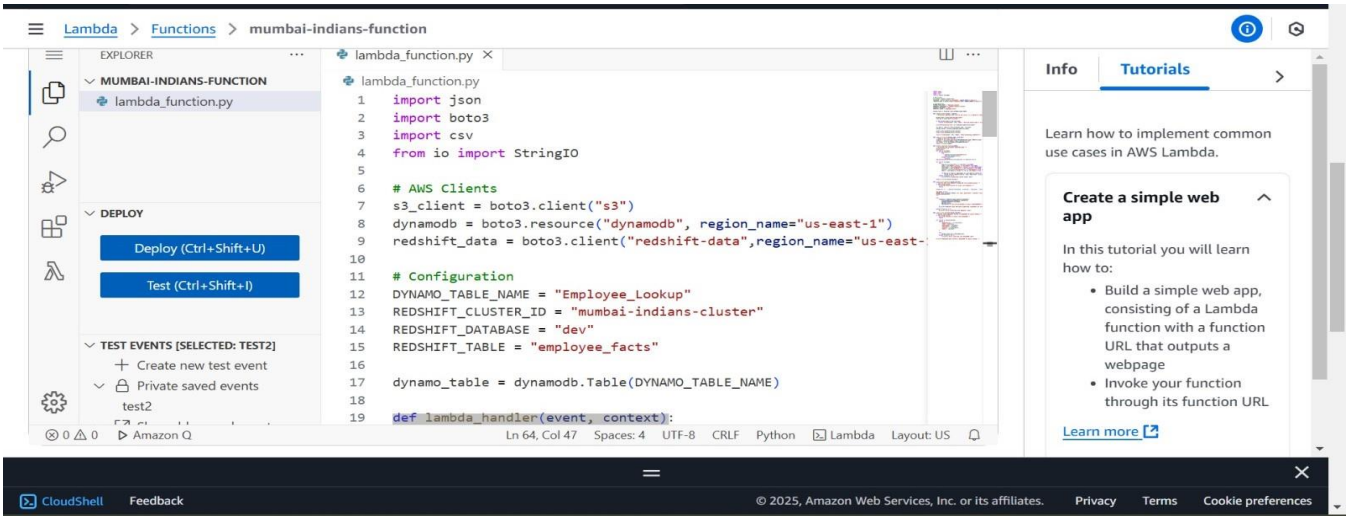


Fig 5: Lambda Function code

Creating the "Mumbai-Indians-Function" in AWS Lambda

To create the "Mumbai-Indians-Function" in AWS Lambda and configure it with the necessary permissions and test case, follow these steps:

Step 1: Create the Lambda Function

1. **Access AWS Lambda**:
   - Log in to the AWS Management Console.
   - Navigate to the Lambda service.
2. **Create a New Function**:
   - Click on "Create function".
   - Choose "Author from scratch".
   - Enter "Mumbai-Indians-Function" as the function name.
   - Select "Python 3.x" as the runtime.
   - Choose an existing role or create a new role with basic Lambda permissions.
3. **Configure the Function**:
   - In the function code section, copy and paste the provided Python code.
   - Click "Deploy" to save the changes.

Step 2: Attach Necessary Permissions

1. **IAM Role Configuration**:
   - Navigate to the IAM service.
   - Find the role associated with your Lambda function.
   - Attach the following policies to the role:
     - AmazonRedshiftFullAccess
     - AmazonS3FullAccess
     - AmazonDynamoDBFullAccess
     - SecretsManagerReadWrite
     - AWSLambdaBasicExecutionRole

Step 3: Create a Test Event

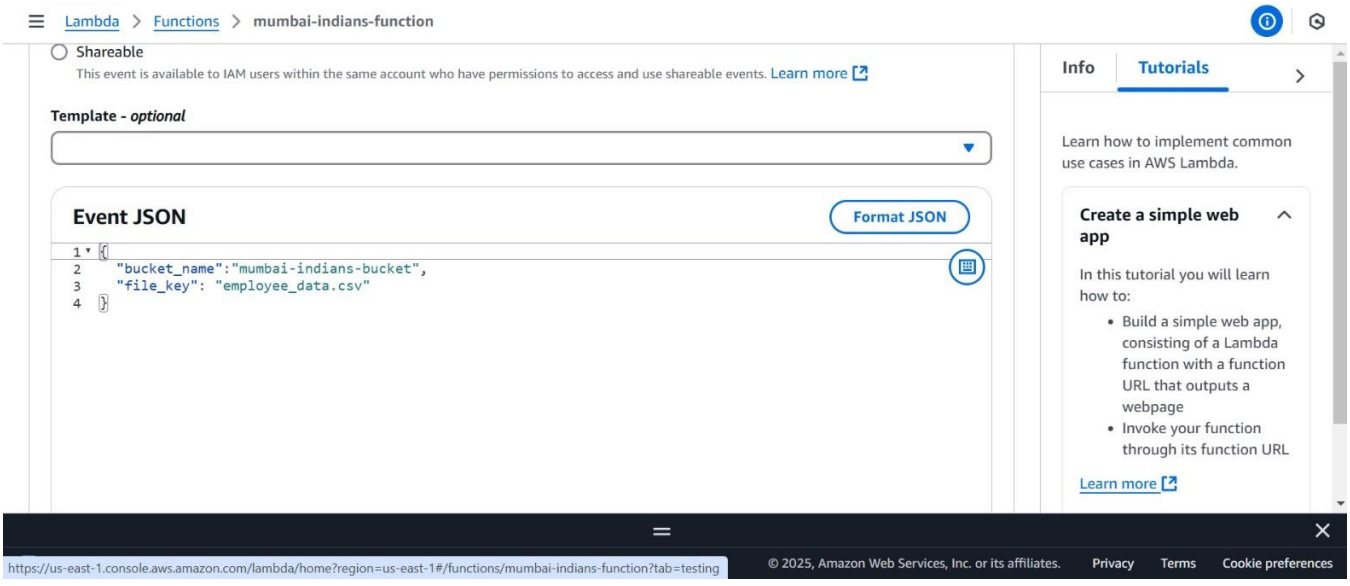


Fig 6: Test format

1. **Configure Test Event**:

- In the Lambda function console, click on "Test".
- Create a new test event with the following JSON format:

```
{
  "bucket_name": "Mumbai-indians-bucket",
  "file_key": "employee_data.csv"
}
```

6. **Save and Run the Test**:
   - Save the test event.
   - Click "Test" to execute the function and verify that it processes the data correctly.

Explanation of the Code

The provided Python code for the Lambda function performs the following tasks:

1. **AWS Clients Initialization**:
   - Initializes clients for S3, DynamoDB, and Redshift Data API.
2. **Configuration**:
   - Defines constants for DynamoDB table name, Redshift cluster ID, database, and table name.
3. **Lambda Handler**:
   - The main function (lambda_handler) processes the event to read the bucket name and file key.
   - Reads the CSV file from S3.
   - Processes and validates the data.
   - Inserts valid records into Redshift and DynamoDB.
4. **Helper Functions**:
   - read_csv_from_s3: Reads a CSV file from S3 and returns a list of dictionaries.
   - process_employee_data: Processes and validates the CSV data.
   - insert_into_redshift: Inserts valid records into Redshift using the Redshift Data API.
   - insert_into_dynamodb: Stores valid records in DynamoDB for quick lookup.

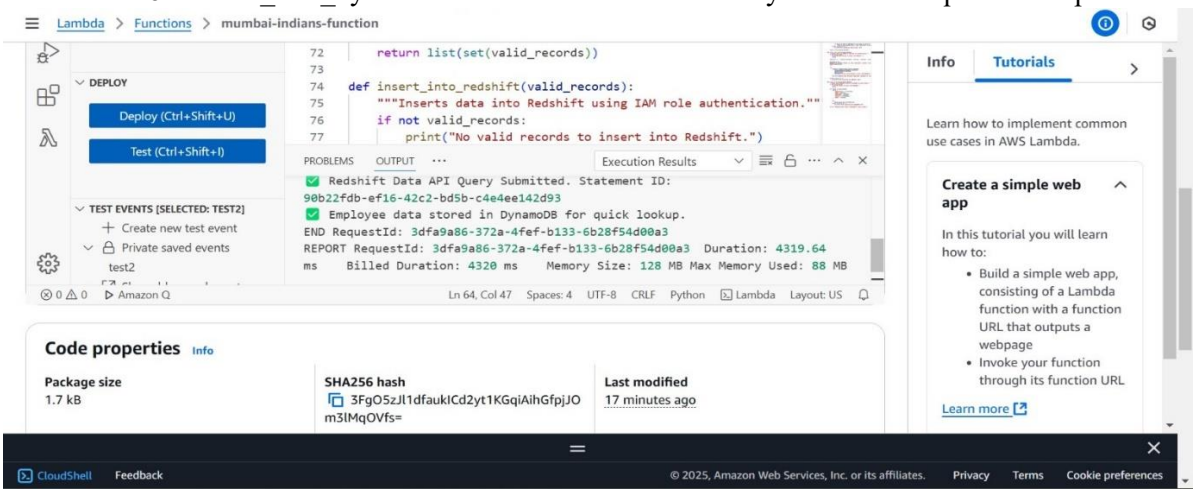


Fig 7: Result

Functional Requirement: 04

- **Generating Redshift Cluster.**

Mumbai-Indians-Cluster Setup and Data Aggregation

Step 1: Creating the Cluster

A new Redshift cluster named **"Mumbai-Indians-Cluster"** was created to manage and analyze the data efficiently. This cluster provides the necessary computational power and storage to handle large datasets.

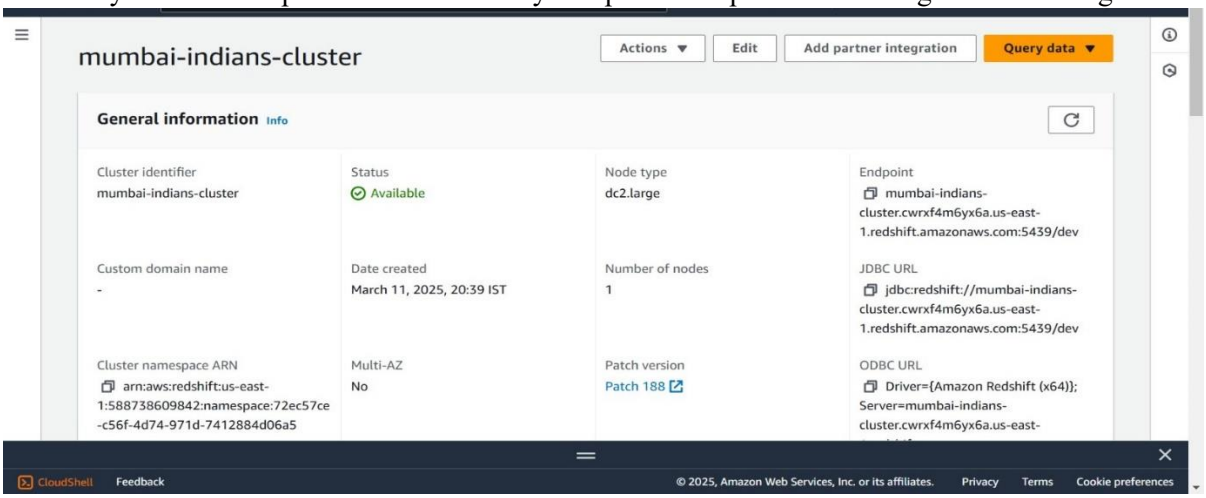


Fig 8: Creating redshift cluster and cluster information

Step 2: Creating Tables in Query Editor

Two tables were created in the Redshift query editor:

1. **Employee_Facts**:
   - **Columns**: employee_id, name, department, join_date, salary, email
   - **Purpose**: Stores detailed information about each employee.
   - **Employee_Fact SQL Query:**

```
CREATE TABLE Employee_Facts (
   Employee_ID INT,
   Name VARCHAR (100),
   Department VARCHAR (50),
   Join_Date DATE,
   Salary INT
   Email VARCHAR (100)
);
```

2. **Department_Summary**:
   - **Columns**: department, total_employees, avg_salary
   - **Purpose**: Stores aggregated data by department, including the total number of employees and the average salary.
   - **Department_Summary Table**

```
CREATE TABLE Department_Summary (
    Department VARCHAR(50),
    Total_Employees INT,
```

```
    Average_Salary INT
);
```

Step 3: Data Aggregation and Insertion

Data from the Employee_Facts table was aggregated by department and inserted into the Department_Summary table. The aggregation included calculating the total number of employees and the average salary for each department.

SQL Query Example:

```
INSERT INTO Department_Summary (department, total_employees, avg_salary)
SELECT department, COUNT(employee_id) AS total_employees, AVG(salary) AS avg_salary
FROM Employee_Facts
GROUP BY department;
```

Results:

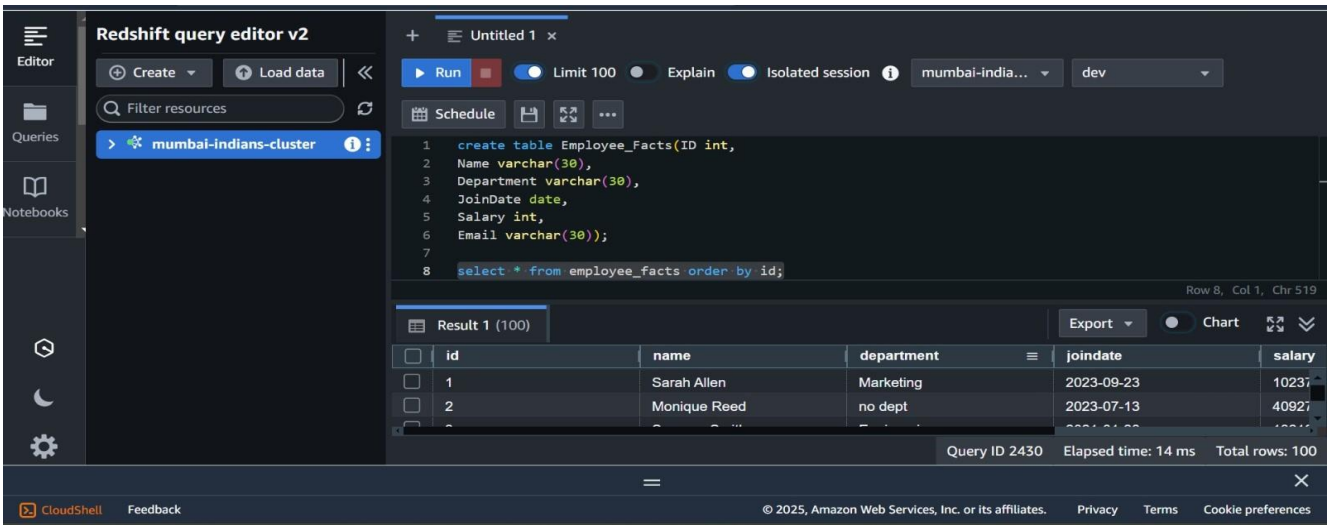


Fig 9: Employee_Facts table

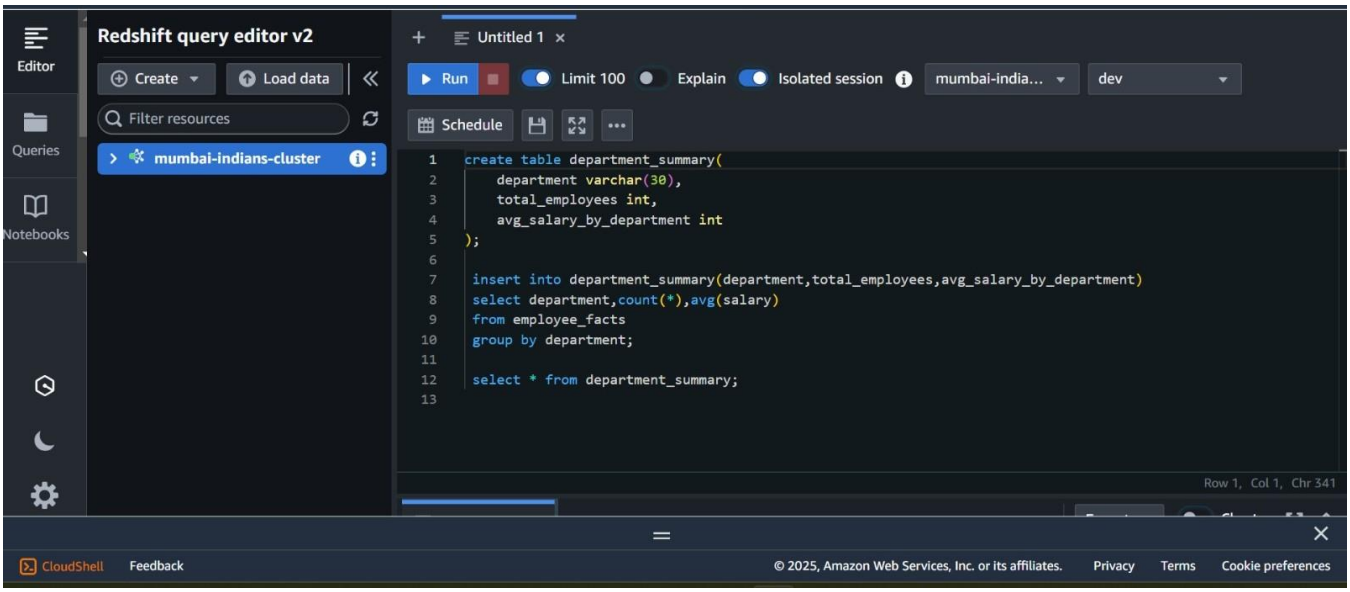


Fig 10: Department_summary table

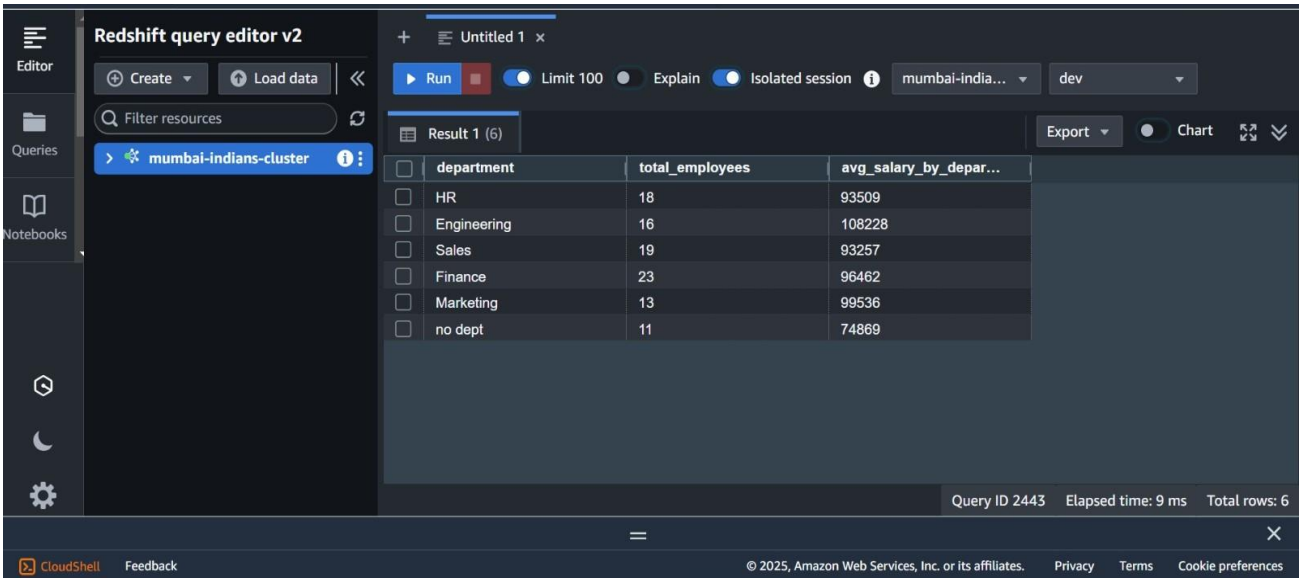


Fig 11: Aggregated result

Functional requirement: 05

- **Generating tables in AWS DynamoDB for quick lookups of the Employee information.**

DynamoDB Table: Employee_Lookup

Table Creation and Configuration

The Employee_Lookup table was created in Amazon DynamoDB to store employee data for quick lookup and efficient querying. The table is configured with the following key attributes:

- Partition Key: Employee_ID
  - This key uniquely identifies each employee record in the table, ensuring efficient data retrieval.

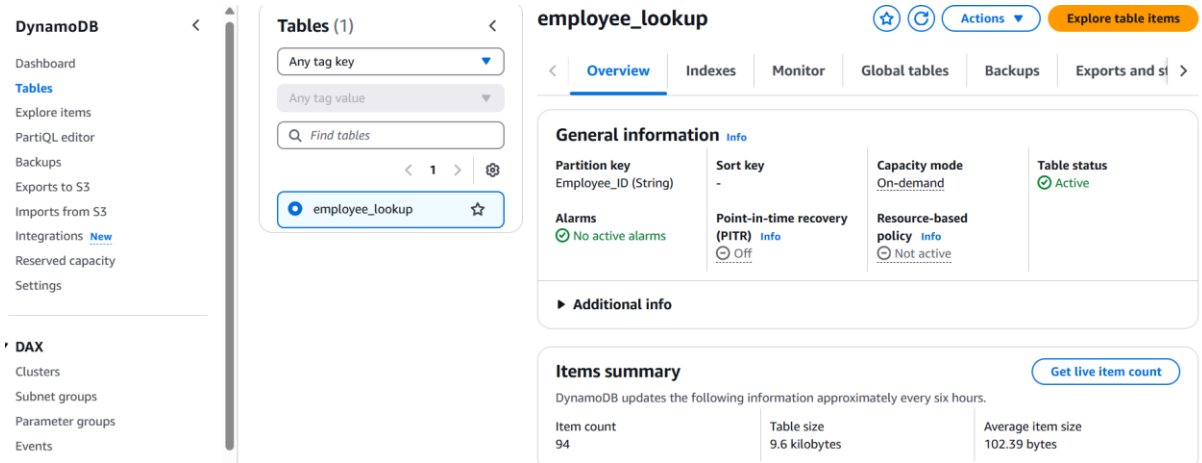


Fig 12: Employee_lookup DynamoDB table

Explore Items

The Explore Items feature in DynamoDB allows you to view and manage the data stored in the table. The screenshots show the following:

1. Table Overview:
    - Displays the table name (Employee_Lookup) and the partition key (Employee_ID).
    - Provides a summary of the table's configuration, including read/write capacity settings and item count.
2. Item Details:
    - Shows individual employee records with attributes such as Employee_ID, name, department, join_date, salary, and email.
    - Allows for easy viewing, editing, and deletion of items directly from the console.

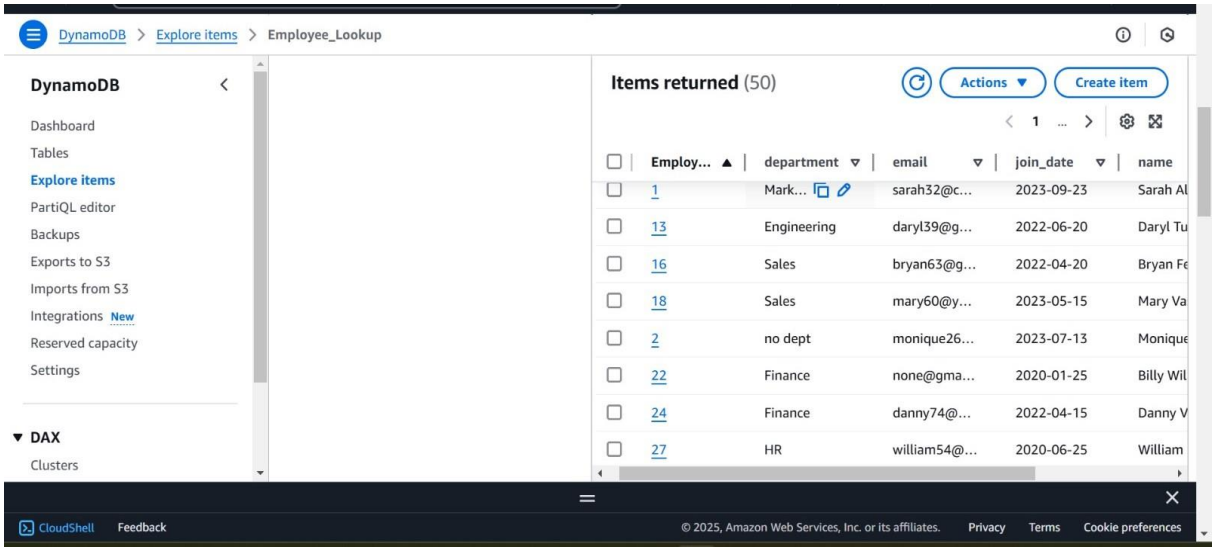


Fig 13: Exploring the table items.

Code Execution

Employee Data Processing Using AWS Lambda, S3, DynamoDB, and Redshift

1. Introduction

This Lambda function processes employee data stored in an AWS S3 bucket, validates the data, and then stores it in Amazon Redshift and DynamoDB for efficient querying. It performs data extraction, transformation, and loading (ETL) operations.

2. Technologies Used

AWS Lambda - Serverless function execution

Amazon S3 - Storage for CSV files

Amazon DynamoDB - Quick lookup storage

Amazon Redshift - Data warehouse for analysis

Boto3 - AWS SDK for Python to interact with AWS services

Python - Core programming language for implementation

3. Workflow Overview

1. The function is triggered with an S3 event containing the `bucket_name` and `file_key` (CSV file path).
2. The CSV file is retrieved from S3.
3. The data is validated (checking for missing values and ensuring correct formats).
4. Transformation:
   - Converts `JoinDate` into the proper date format for Redshift (`YYYY-MM-DD`).
   - Replaces null emails with a default value (`unknown@example.com`).
   - Removes duplicate records before inserting.
5. Stores the cleaned data in:
   - Amazon Redshift (for analysis)
   - Amazon DynamoDB (for quick lookups)

4. Code Breakdown

4.1. AWS Clients and Configuration

```python
import json
import boto3
import csv
from io import StringIO

# AWS Clients
s3_client = boto3.client("s3")
dynamodb = boto3.resource("dynamodb", region_name="us-east-1")
redshift_data = boto3.client("redshift-data", region_name="us-east-1")

# Configuration
DYNAMO_TABLE_NAME = "employee_lookup"
REDSHIFT_CLUSTER_ID = "mumbai-indians-cluster"
REDSHIFT_DATABASE = "dev"
REDSHIFT_TABLE = "Employee_Facts"
```

- Initializes S3, DynamoDB, and Redshift clients.
- Stores table and cluster configurations.

4.2. Lambda Entry Function

```python
def lambda_handler(event, context):
    """Processes employee data from S3 and stores it in Redshift & DynamoDB."""

    bucket_name = event.get("bucket_name")
    file_key = event.get("file_key")

    if not bucket_name or not file_key:
        return {"statusCode": 400, "body": "Missing bucket_name or file_key in event"}

    print(f"Processing file: s3://{bucket_name}/{file_key}")

    csv_data = read_csv_from_s3(bucket_name, file_key)
```

```
    valid_records = process_employee_data(csv_data)

    insert_into_redshift(valid_records)
    insert_into_dynamodb(valid_records)

    return {"statusCode": 200, "body": "Data processing complete!"}
```
- Receives S3 event data.
- Reads the CSV file from S3.
- Processes and validates the data.
- Inserts the cleaned data into Redshift and DynamoDB.

4.3. Read CSV File from S3
```python
def read_csv_from_s3(bucket_name, file_key):
    """Reads a CSV file from S3."""
    response = s3_client.get_object(Bucket=bucket_name, Key=file_key)
    content = response["Body"].read().decode("utf-8")
    csv_reader = csv.DictReader(StringIO(content))
    return list(csv_reader)
```
- Fetches CSV file from S3 and parses it.

4.4. Data Processing and Transformation
```python
def process_employee_data(csv_data):
    """Processes and validates CSV data."""
    valid_records = []
    salaries = []

    # First pass to collect all salaries
    for row in csv_data:
        if row["Salary"]:
            try:
```

```
            salary = int(float(row["Salary"]))
            salaries.append(salary)
        except ValueError:
            continue

    # Calculate average salary
    avg_salary = sum(salaries) / len(salaries) if salaries else 0

    for row in csv_data:
        try:
            emp_id = int(row["ID"]) if row["ID"] else None
            name = row["Name"].strip() if row["Name"] else None
            department = row["Department"].strip() if row["Department"] else 'no dept'
            join_date = row["JoinDate"] if row["JoinDate"] else None
            salary = int(float(row["Salary"])) if row["Salary"] else avg_salary
            email = row["Email"].strip() if "@" in row["Email"] else 'none@gmail.com'

            if emp_id and name and department and join_date and salary and email:
                valid_records.append((emp_id, name, department, join_date, salary, email))
        except Exception as e:
            print(f"Error processing record {row}: {e}")

    return list(set(valid_records))
```

Transformations applied:

- ☑ Replaces missing deoartments with no dept
- ☑ Replaces missing emails with `"none@gmail.com"`.
- ☑ Removes duplicate records using a set.
- ☑ Replaces missing salaries with avg_salary

4.5. Insert Data into Redshift

```python
def insert_into_redshift(valid_records):
    if not valid_records:
        print("No valid records to insert into Redshift.")
        return
```

```
    values_str = ", ".join([f"({rec[0]}, '{rec[1]}', '{rec[2]}', '{rec[3]}', {rec[4]}, '{rec[5]}')" for rec in
valid_records])

    sql_query = f"""
    INSERT INTO {REDSHIFT_TABLE} (employee_id, name, department, join_date, salary, email)
    VALUES {values_str};
    """

    try:
        response = redshift_data.execute_statement(
            ClusterIdentifier=REDSHIFT_CLUSTER_ID,
            Database=REDSHIFT_DATABASE,
            Sql=sql_query,
            SecretArn="arn:aws:secretsmanager:us-east-1:481665088600:secret:redshift!redshift-cluster-2-
awsuser-tUlHaK
        )
        print(f"☑ Redshift Data API Query Submitted. Statement ID: {response['Id']}")

    except Exception as e:
        print(f"❌ Error inserting into Redshift: {e}")
```

- Inserts transformed data into Redshift.
- Uses Secrets manager arn to connect to redshift.

4.6. Insert Data into DynamoDB

```python
def insert_into_dynamodb(valid_records):
    """Stores valid employee records in DynamoDB for quick lookup."""
    if not valid_records:
        print("No records to insert into DynamoDB.")
        return

    for record in valid_records:
        item = {
            "employee_id": str(record[0]),
            "name": record[1],
```

```
            "department": record[2],
            "join_date": record[3],
            "salary": str(record[4]),
            "email": record[5]
        }

        try:
            dynamo_table.put_item(Item=item)
        except Exception as e:
            print(f"❌ Error inserting into DynamoDB: {e}")

    print("☑ Employee data stored in DynamoDB for quick lookup.")
```

- Stores data in DynamoDB for fast lookups.

---

5. Conclusion

This Lambda function performs ETL operations by extracting employee data from S3, transforming it (removing duplicates, formatting dates, filling missing values), and then loading it into Redshift (for analytics) and DynamoDB (for quick lookups). The optimized workflow ensures accurate and efficient data processing.

Conclusion

The Employee Data Management System project successfully delivers a comprehensive solution for managing employee records within an organization. By leveraging a suite of AWS services and other tools, the system ensures efficient processing, storage, and retrieval of employee data. The project encompasses the entire data lifecycle, from extraction and transformation to loading, quick lookups, and archival. Throughout the project, we have meticulously implemented SQL queries, validated and cleaned data, and enriched it to enhance its value. The integration of AWS S3, Redshift, DynamoDB, and Teradata provides a robust infrastructure that supports scalability, data integrity, and long-term storage. The automation of data processing reduces manual effort and minimizes errors. The system can handle large volumes of data and scale as needed (scalability) and is quickly accessible to employee information through DynamoDB.