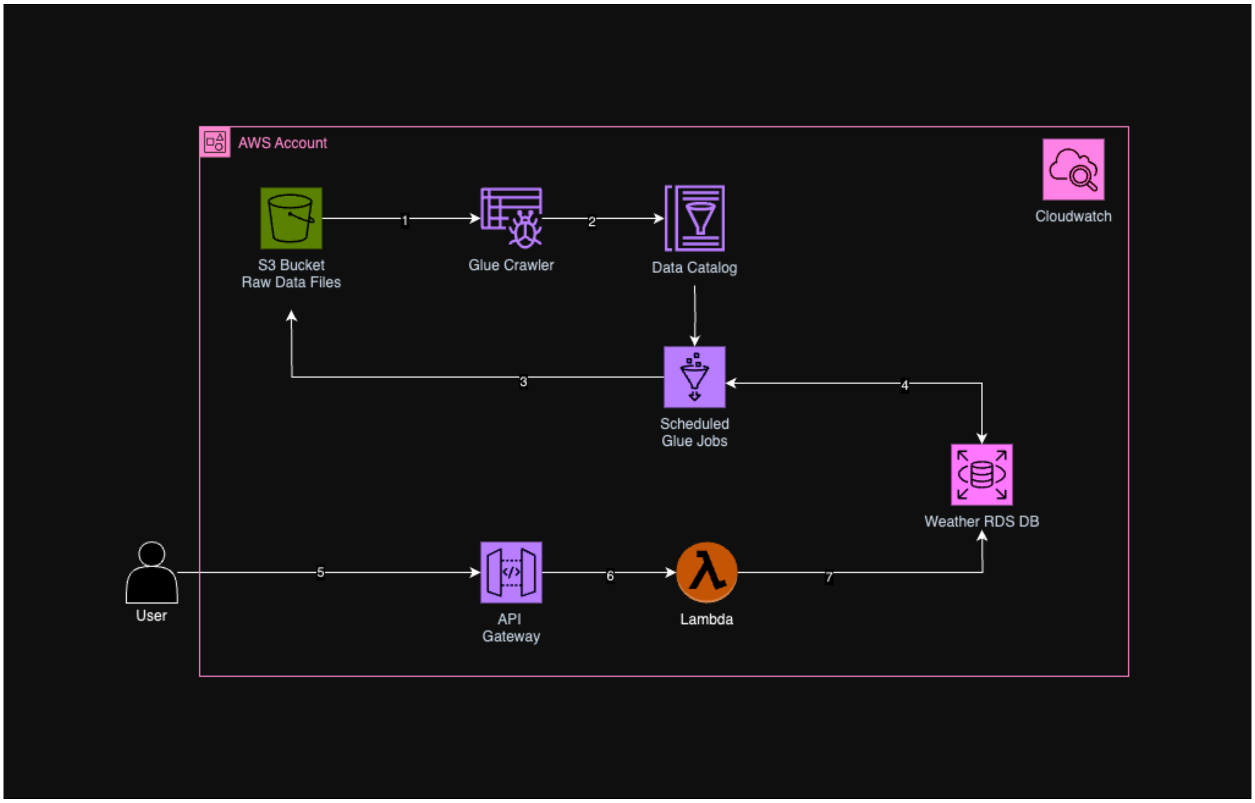
Deployment Summary for Weather Data Processing on AWS

This document outlines the approach to deploying a weather data processing system using various AWS services. The deployment includes setting up the database, ingesting and transforming data, and creating an API for data access. The key AWS services used are Amazon RDS, AWS Glue, Amazon S3, AWS Lambda, and Amazon API Gateway.



1. Database Setup: Amazon RDS
2. Data Ingestion and Transformation: AWS Glue
3. API Deployment: AWS Lambda and API Gateway
4. Automation with AWS CloudFormation

1. Database Setup: Amazon RDS

Service: Amazon RDS

Steps:

- Create RDS Instance: Launch an Aurora PostgreSQL cluster.
- Configure Security: Set up security groups and IAM roles to control access.

- Create Tables: Use an AWS Lambda function as a CloudFormation custom resource to create the necessary tables in the RDS database.

Lambda Function Code (to create tables in RDS):

```python
import os
import psycopg2
import boto3
import cfnresponse

def handler(event, context):
    response_data = {}
    try:
        conn = psycopg2.connect(
            host=os.getenv('DB_HOST'),
            dbname=os.getenv('DB_NAME'),
            user=os.getenv('DB_USER'),
            password=os.getenv('DB_PASSWORD'),
            port=os.getenv('DB_PORT')
        )
        cursor = conn.cursor()
        cursor.execute('''
        CREATE TABLE IF NOT EXISTS weather_data (
            station_id TEXT,
            date DATE,
            max_temp INTEGER,
            min_temp INTEGER,
            precipitation INTEGER,
            PRIMARY KEY (station_id, date)
        );
        CREATE TABLE IF NOT EXISTS weather_stats (
            station_id TEXT,
            year INTEGER,
            avg_max_temp REAL,
            avg_min_temp REAL,
            total_precipitation REAL,
            PRIMARY KEY (station_id, year)
        );
        ''')
        conn.commit()
        cursor.close()
        conn.close()
        cfnresponse.send(event, context, cfnresponse.SUCCESS, response_data)
    except Exception as e:
        response_data['Error'] = str(e)
        cfnresponse.send(event, context, cfnresponse.FAILED, response_data)
```

2. Data Ingestion and Transformation: AWS Glue

Service: AWS Glue

Steps:

- Create Glue Job: Write a Glue job to read data from S3, ingest it into RDS, and calculate weather statistics.

- Glue Job Script (Python - PySpark):

```python
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

# Initialize Glue context and job
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)

# Define source and target details
s3_source = 's3://your-bucket-name/wx_data/'
rds_jdbc_url = 'jdbc:postgresql://your-rds-endpoint:5432/yourdbname'
rds_user = 'yourusername'
rds_password = 'yourpassword'

# Step 1: Read data from S3
datasource0 = glueContext.create_dynamic_frame.from_options(
    connection_type="s3",
    connection_options={"paths": [s3_source]},
    format="csv",
    format_options={"withHeader": True, "separator": "\t"}
)
```

```
# Step 2: Transform data (handle missing values and convert data types)
applymapping1 = ApplyMapping.apply(
    frame=datasource0,
    mappings=[
        ("col0", "string", "station_id", "string"),
        ("col1", "string", "date", "string"),
        ("col2", "int", "max_temp", "int"),
        ("col3", "int", "min_temp", "int"),
        ("col4", "int", "precipitation", "int")
    ]
)

# Step 3: Write data to RDS
glueContext.write_dynamic_frame.from_options(
    frame=applymapping1,
    connection_type="jdbc",
    connection_options={
        "url": rds_jdbc_url,
        "dbtable": "public.weather_data",
        "user": rds_user,
        "password": rds_password
    }
)

# Step 4: Read ingested data from RDS
datasource1 = glueContext.create_dynamic_frame.from_options(
    connection_type="jdbc",
    connection_options={
        "url": rds_jdbc_url,
        "dbtable": "public.weather_data",
        "user": rds_user,
        "password": rds_password
    }
)

# Convert to DataFrame for transformation
datasource1_df = datasource1.toDF()
datasource1_df.createOrReplaceTempView("weather_data")

# Step 5: Calculate weather statistics
stats_df = spark.sql("""
SELECT
    station_id,
    YEAR(TO_DATE(date, 'YYYYMMDD')) AS year,
    AVG(CASE WHEN max_temp != -9999 THEN max_temp / 10.0 ELSE NULL END) AS avg_max_temp,
    AVG(CASE WHEN min_temp != -9999 THEN min_temp / 10.0 ELSE NULL END) AS avg_min_temp,
    SUM(CASE WHEN precipitation != -9999 THEN precipitation / 10.0 ELSE 0 END) AS total_precipitation
FROM weather_data
GROUP BY station_id, year
""")

# Convert back to DynamicFrame
stats_dynamic_frame = DynamicFrame.fromDF(stats_df, glueContext, "stats_dynamic_frame")

# Step 6: Write statistics data to RDS
glueContext.write_dynamic_frame.from_options(
    frame=stats_dynamic_frame,
    connection_type="jdbc",
    connection_options={
        "url": rds_jdbc_url,
        "dbtable": "public.weather_stats",
        "user": rds_user,
        "password": rds_password
    }
)

# Commit the job
job.commit()
```

- Schedule Glue Job: Use Amazon EventBridge to schedule the Glue job to run daily.

EventBridge Rule:

```bash
aws events put-rule --schedule-expression 'rate(1 day)' --name DailyIngestionTransformationRule
aws events put-targets --rule DailyIngestionTransformationRule --targets "Id"="1","Arn"="arn:aws:glue:us-west-2:your-account-id:job/weather-ingestion-transformation-job"
```

3. API Deployment: AWS Lambda and API Gateway

Service: AWS Lambda and Amazon API Gateway

Steps:

- Create Lambda Functions: To handle API requests for querying weather data and statistics, Create Lambda functions.

Lambda Function Code:

```python
import json
import os
import psycopg2
from psycopg2.extras import RealDictCursor

def get_db_connection():
    conn = psycopg2.connect(
        host=os.getenv('DB_HOST'),
        dbname=os.getenv('DB_NAME'),
        user=os.getenv('DB_USER'),
        password=os.getenv('DB_PASSWORD'),
        port=os.getenv('DB_PORT')
    )
    return conn

def query_weather_data(event, context):
    station_id = event.get('queryStringParameters', {}).get('station_id')
    start_date = event.get('queryStringParameters', {}).get('start_date')
    end_date = event.get('queryStringParameters', {}).get('end_date')

    query = "SELECT * FROM weather_data WHERE 1=1"
    params = []
    if station_id:
        query += " AND station_id = %s"
        params.append(station_id)
    if start_date:
        query += " AND date >= %s"
        params.append(start_date)
    if end_date:
        query += " AND date <= %s"
        params.append(end_date)

    conn = get_db_connection()
    cursor = conn.cursor(cursor_factory=RealDictCursor)
    cursor.execute(query, params)
    results = cursor.fetchall()
    cursor.close()
    conn.close()

    return {
        'statusCode': 200,
        'body': json.dumps(results, default=str)
    }

def query_weather_stats(event, context):
    station_id = event.get('queryStringParameters', {}).get('station_id')
    year = event.get('queryStringParameters', {}).get('year')

    query = "SELECT * FROM weather_stats WHERE 1=1"
    params = []
    if station_id:
        query += " AND station_id = %s"
        params.append(station_id)
    if year:
        query += " AND year = %s"
        params.append(year)

    conn = get_db_connection()
    cursor = conn.cursor(cursor_factory=RealDictCursor)
    cursor.execute(query, params)
    results = cursor.fetchall()
    cursor.close()
    conn.close()

    return {
        'statusCode': 200,
        'body': json.dumps(results, default=str)
    }
```

Package Lambda Functions: Package the Lambda function code and dependencies into a zip file and deploy using AWS CLI.

Packaging:

```bash
mkdir lambda_function
cd lambda_function
echo "psycopg2-binary" > requirements.txt
pip install -r requirements.txt -t .
zip -r9 ../lambda_function.zip .
```

Deploy Lambda Functions:

```bash
aws lambda create-function --function-name QueryWeatherData \
    --zip-file fileb://../lambda_function.zip --handler lambda_function.query_weather_data \
    --runtime python3.8 --role arn:aws:iam::your

-account-id:role/your-lambda-role \
    --environment
Variables="{DB_HOST=your-rds-endpoint,DB_NAME=yourdbname,DB_USER=yourusername,
DB_PASSWORD=yourpassword,DB_PORT=5432}"

aws lambda create-function --function-name QueryWeatherStats \
    --zip-file fileb://../lambda_function.zip --handler lambda_function.query_weather_stats \
    --runtime python3.8 --role arn:aws:iam::your-account-id:role/your-lambda-role \
    --environment
Variables="{DB_HOST=your-rds-endpoint,DB_NAME=yourdbname,DB_USER=yourusername,
DB_PASSWORD=yourpassword,DB_PORT=5432}"
```

- Set Up API Gateway: Configure API Gateway to expose the Lambda functions as REST API endpoints.

API Gateway Configuration:

```bash
aws apigateway create-rest-api --name "Weather API"
API_ID=$(aws apigateway get-rest-apis --query 'items[?name==`Weather API`].id' --output text)

# Create resources and methods for /weather
PARENT_RESOURCE_ID=$(aws apigateway get-resources --rest-api-id $API_ID --query
'items[?path==`/`].id' --output text)
WEATHER_RESOURCE_ID=$(aws apigateway create-resource --rest-api-id $API_ID
--parent-id $PARENT_RESOURCE_ID --path-part "weather" --query 'id' --output text)

aws apigateway put-method --rest-api-id $API_ID --resource-id $WEATHER_RESOURCE_ID
--http-method GET --authorization-type "NONE"
aws apigateway put-integration --rest-api-id $API_ID --resource-id
$WEATHER_RESOURCE_ID --http-method GET --type AWS_PROXY --integration-http-method
POST --uri
"arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/arn:aws:lambda:us-west-2:yo
ur-account-id:function:QueryWeatherData/invocations"

# Create resources and methods for /weather/stats
```

```
WEATHER_STATS_RESOURCE_ID=$(aws apigateway create-resource --rest-api-id $API_ID --parent-id $PARENT_RESOURCE_ID --path-part "weather/stats" --query 'id' --output text)

aws apigateway put-method --rest-api-id $API_ID --resource-id $WEATHER_STATS_RESOURCE_ID --http-method GET --authorization-type "NONE"
aws apigateway put-integration --rest-api-id $API_ID --resource-id $WEATHER_STATS_RESOURCE_ID --http-method GET --type AWS_PROXY --integration-http-method POST --uri "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/arn:aws:lambda:us-west-2:your-account-id:function:QueryWeatherStats/invocations"
```

4. Automation with AWS CloudFormation

Service: AWS CloudFormation

Steps:
- Create CloudFormation Template: Write a CloudFormation template to automate the provisioning of all resources.

Example CloudFormation Template (`cloudformation.yaml`):

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  WeatherDataBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: your-weather-data-bucket

  WeatherRDSCluster:
    Type: AWS::RDS::DBCluster
    Properties:
      Engine: aurora-postgresql
      MasterUsername: !Ref DBUsername
      MasterUserPassword: !Ref DBPassword
      BackupRetentionPeriod: 7
      DBSubnetGroupName: your-subnet-group
      VpcSecurityGroupIds:
        - sg-xxxxxxx
    DeletionPolicy: Snapshot

  WeatherRDSInstance:
    Type: AWS::RDS::DBInstance
    Properties:
      DBClusterIdentifier: !Ref WeatherRDSCluster
      DBInstanceClass: db.r5.large

  CreateRDSTablesFunction:
    Type: AWS::Lambda::Function
    Properties:
      Handler: lambda_function.handler
      Role: arn:aws:iam::your-account-id:role/your-lambda-role
      Code:
        S3Bucket: your-bucket-name
        S3Key: function.zip
      Runtime: python3.8
      Environment:
        Variables:
          DB_HOST: !GetAtt WeatherRDSCluster.Endpoint.Address
          DB_NAME: yourdbname
          DB_USER: yourusername
          DB_PASSWORD: yourpassword
          DB_PORT: '5432'

  CreateRDSTablesInvoke:
    Type: Custom::CreateRDSTables
    Properties:
      ServiceToken: !GetAtt CreateRDSTablesFunction.Arn

  WeatherGlueRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service: glue.amazonaws.com
            Action: sts:AssumeRole
      Policies:
        - PolicyName: GlueS3Access
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - s3:ListBucket
                  - s3:GetObject
                Resource:
                  - arn:aws:s3:::your-weather-data-bucket
                  - arn:aws:s3:::your-weather-data-bucket/*

  WeatherIngestionTransformationJob:
    Type: AWS::Glue::Job
    Properties:
      Name: weather-ingestion-transformation-job
      Role: !GetAtt WeatherGlueRole.Arn
      Command:
        Name: glueetl
        ScriptLocation: s3://your-script-bucket/glue-scripts/weather_ingestion_transformation.py
        PythonVersion: 3
      DefaultArguments:
        --job-language: python
        --extra-py-files: s3://your-script-bucket/glue-scripts/your_additional_modules.zip
      MaxCapacity: 4

  WeatherIngestionTransformationTrigger:
    Type: AWS::Events::Rule
    Properties:
      ScheduleExpression: rate(1 day)
      Targets:
        - Arn: !GetAtt WeatherIngestionTransformationJob.Arn
          Id: "1"
```