RDS DATABASE DRIFT DETECTION : LAMBDA FUNCTION

Standard Document – Runbook

This document has been meticulously prepared by *Prashant Dixit (*CBTS – ONX) to address the specific requirements and inquiries of Third Bank, represented by *Michael Pento* and *Billy Sundararajan*. With a commitment to delivering tailored solutions and comprehensive insights, this document aims to provide description about the ‘DRIFT DETECTION’ Lambda function which us used to identify drifts in RDS PostgreSQL databases present in different AWS Organization accounts.

The document can be easily referred as a standard runbook for this Lambda Function.

**Table of Contents:**

[1. Purpose of the function 3](#_Toc150290055)

[2. Code Breakup/Walkthrough 3](#_Toc150290056)

[3. CIS Statements Used & Purpose 8](#_Toc150290057)

[4. How to call the function? 18](#_Toc150290058)

[5. Monitoring & Debugging 23](#_Toc150290059)

[6. Add new reports to the Code. 24](#_Toc150290060)

[7. Lambda Function 2 : One to cover only new patterns ro/rw/BigID/Owner formats. 26](#_Toc150290061)

# Purpose of the function

The code is used to detect drifts (permissions, roles, parameters etc.) in RDS PostgreSQL databases to fulfil CIS requirements and to perform regular audits to understand deviations from the information security standards.

This Lambda function serves as an essential automation tool for monitoring and assessing the connectivity status of Amazon RDS instances across multiple AWS accounts. By assuming roles, retrieving data, and checking secret management, it provides valuable insights into the health and configuration of these critical database resources. The generated summary table offers a concise overview of the function's actions, facilitating efficient tracking and management of RDS instances.

# Code Breakup/Walkthrough

The code is used via an AWS Lambda Function, or a lambda expression written in Python language to anonymously define single user single/multi use functions.

1. Code/Repo Location

Below mentioned are the GitHub Code repositories that you can use to deploy the lambda functions via Terraform.

Dev Repo : <https://github.info53.com/fitb-edm-udb/aws-sc-tf2cf/tree/feature/prashant/dev/drift_detection_new_test>

Sandbox Repo : <https://github.info53.com/fitb-edm-udb/aws-sc-tf2cf/tree/feature/prashant/sandbox/drift_detection_new_test>

Production Repo :

<https://github.info53.com/fitb-edm-udb/aws-sc-tf2cf/tree/feature/prashant/prod/drift_detection_new_test>

Source Repo used :

<https://github.info53.com/fitb-edm-dba/adithya_lambdas/tree/feature/create_lambda>

1. AWS Services Used

The function used multiple AWS services and features. Here's a breakdown of the AWS services and features used in the code:

**Amazon RDS (Relational Database Service)**:

* + Amazon RDS is used for managing and running relational databases. The code connects to RDS instances and performs SQL queries to retrieve information about database objects and permissions.

**AWS Secrets Manager**:

* + AWS Secrets Manager is used for storing and managing sensitive information, such as database connection credentials. The code retrieves database connection secrets from Secrets Manager to connect to RDS instances securely.

**AWS Lambda**:

* + AWS Lambda is not directly used in the code, but it's a serverless compute service that can execute code in response to events. Lambda functions can be used to trigger the code you provided in response to specific events or on a schedule.

**Amazon S3 (Simple Storage Service)**:

* + Amazon S3 is not explicitly mentioned in the provided code, but it is commonly used to store and manage files, including logs, reports, or backups generated by Lambda functions or other AWS services.

**AWS IAM (Identity and Access Management)**:

* + AWS IAM is used to manage access control and permissions for AWS resources. In the code, roles and permissions for accessing AWS services and resources are managed using IAM roles.

1. Libraries/Modules Used

The code uses multiple python libraries and modules. Each of them are explained below.

**import boto3**: This imports the **boto3** library, which is the official AWS SDK for Python. It allows you to interact with various AWS services programmatically.

**import botocore**: This imports the **botocore** library, which is a low-level library for handling AWS interactions. It is often used internally by **boto3**.

**from botocore.exceptions import ClientError**: This imports the **ClientError** class from the **botocore.exceptions** module. **ClientError** is used for handling exceptions related to AWS service clients.

**import postgres\_utils as pgs**: This imports a module named **postgres\_utils** and gives it the alias **pgs**. It suggests that there's a custom Python module called **postgres\_utils** that contains utility functions related to PostgreSQL or database operations.

**from postgres\_utils import pg**: This imports a specific object or function named **pg** from the **postgres\_utils** module. It's used to access a particular function or class from the custom module.

**from csv2pdf import convert**: This imports a function named **convert** from a module named **csv2pdf**. This suggests that there's a custom module called **csv2pdf** that contains a function for converting CSV data to PDF format.

**from PIL import ImageFont**: This imports the **ImageFont** class from the Python Imaging Library (PIL), which is often used for working with images and fonts in image processing tasks.

**import logging**: This imports the **logging** module, which is used for adding logging capabilities to the Python script. It allows you to record and manage log messages.

**from datetime import datetime**: This imports the **datetime** class from the Python **datetime** module. It's used for working with date and time values.

**import csv**: This imports the built-in **csv** module, which provides functionality for working with CSV (Comma-Separated Values) files.

**import os**: This imports the built-in **os** module, which is used for interacting with the operating system. It allows you to perform file and directory operations.

**import pgdb**: This imports the **pgdb** module, which is often used for working with PostgreSQL databases in Python. It provides a database interface for PostgreSQL.

1. Functions used and overview of the code blocks:

Code defines a set of functions used for interacting with AWS services and handling RDS (Relational Database Service) instances and snapshots. Here's the purpose of each function and an overview of what each block of code does:

**get\_assume\_role\_session(sts\_role)**: This function obtains temporary AWS credentials by assuming an IAM role (sts\_role) using the Security Token Service (STS) client. It returns the temporary credentials.

**run\_query(secrets\_client, secret\_arn, sql)**: This function runs an SQL query on an RDS instance. It first attempts to establish a database connection using **pgs.get\_connection** from a custom module and secret information retrieved from AWS Secrets Manager. If the connection is successful, it executes the SQL query using **pgs.run\_query\_using\_secrets**. It logs status messages and returns a status code.

**update\_table(rds\_list, parameter\_list, table\_name)**: This function is responsible for updating a table (**table\_name**) with data from a list of RDS instances (**rds\_list**) using an SQL **INSERT INTO** statement. It constructs the SQL query dynamically based on the provided data.

**create\_or\_alter\_table(parameter\_list, table\_name)**: This function creates or alters a table (**table\_name**) in an RDS database. It generates SQL statements to create or alter the table structure based on the list of column headers (**parameter\_list**).

**parse\_instances\_info(rds\_client, acct\_id, acct\_ids\_dict, scan\_id)**: This function retrieves and parses information about RDS instances from target AWS accounts. It uses the **rds\_client** to describe RDS instances and clusters and populates the **rds\_list** with instance details. It also handles exceptions and logs errors. After retrieving the instance information, it creates or updates an "instances\_info" table.

**parse\_snapshots\_info(rds\_client, acct\_id, acct\_ids\_dict, scan\_id)**: This function retrieves and parses information about RDS snapshots from target AWS accounts. It uses the **rds\_client** to describe RDS instance snapshots and populates the **rds\_snapshots\_list** with snapshot details. It also handles exceptions and logs errors. After retrieving the snapshot information, it creates or updates a "snapshots\_info" table.

**get\_secrets\_list(secrets\_client)**: This function retrieves a list of secrets from AWS Secrets Manager using the provided **secrets\_client**. It creates a dictionary mapping secret names to their ARNs.

**get\_account\_ids()**: This function retrieves a list of AWS account IDs based on the environment and account IDs specified in environment variables. It uses the **sts\_org\_arn** to assume an IAM role in AWS Organizations to list accounts. It filters accounts based on the environment and specified account IDs and returns a dictionary mapping account IDs to account names.

1. SQL Based Functions:

For each function, the general pattern is to iterate through the result list, convert each row into a dictionary using the each.\_asdict() method (assuming each is a namedtuple), and append the resulting dictionary to the appropriate list with predefined keys. The function then returns the updated list.

**get\_user\_roles\_list**: This function takes four arguments: **scan\_id**, **acct\_id**, **user\_roles\_list**, and **result**. It iterates through the **result** list, which presumably contains namedtuples (or similar objects with named attributes), converts each result row into a dictionary, and appends it to the **user\_roles\_list** as a dictionary with specific keys.

**get\_database\_permissions\_list**: Similar to the first function, this one parses database permission-related results and appends them to the **database\_permissions\_list**.

**get\_schema\_privs\_role\_list**: This function processes schema privilege-related results and adds them to the **schema\_privs\_role\_list**.

**get\_role\_specific\_privs\_list**: Handles role-specific privilege results and appends them to the **role\_specific\_privs\_list**.

**get\_role\_priv\_tables\_list** and **get\_role\_specific\_priv\_tables\_list**: These functions deal with privileges related to tables and specific tables, respectively, and store the data in **role\_priv\_tables\_list** and **role\_specific\_priv\_tables\_list**.

**get\_views\_ownership\_usage\_privs\_list**: Parses views, ownership, and usage privilege results, and stores them in **views\_ownership\_usage\_privs\_list**.

**get\_view\_privs\_role\_list**: Handles view privilege-related results and adds them to **view\_privs\_role\_list**.

**get\_sequence\_ownership\_usage\_privs\_list**: This function processes sequences, ownership, and usage privilege results and stores them in **sequence\_ownership\_usage\_privs\_list**.

**get\_roles\_specific\_privileges\_sequences\_list**: Deals with specific privilege sequences and appends them to **roles\_specific\_privileges\_sequences\_list**.

**get\_roles\_privs\_fdw\_list**, **get\_roles\_login\_fdw\_list**, and **get\_roles\_privs\_language\_list**: These functions handle foreign data wrapper (FDW) permissions, login permissions for FDWs, and language privileges, respectively.

**get\_functions\_ownership\_roles\_list**: This function processes functions, ownership, and roles related to them.

**get\_audit\_roles\_list**: Parses audit roles data and appends it to **audit\_roles\_list**.

**get\_public\_roles\_list**: Processes public roles data and appends it to **public\_roles\_list**. This function appears to handle different cases based on the number of keys in the dictionary, suggesting varying result formats.

1. Report Generation Core Function

Python function, **generate\_csv\_and\_pdf\_reports\_for\_the\_drift\_tables** is an important function whichperforms several tasks related to generating and storing CSV and PDF reports based on data from various database tables. This function connects to a reporting database, retrieves data from specified tables for a given scan, generates CSV and PDF reports, and uploads these reports to an S3 bucket for storage. It processes multiple tables, handling different structures and default parameters as needed for each table.

**Function Signature and Parameters**:

**secrets\_client**: A client object used to connect to a database with secrets.

**reporting\_db\_secret\_arn**: ARN (Amazon Resource Name) of the secret used to access the reporting database.

**table\_names\_list**: A list of names of the tables for which reports need to be generated.

**scan\_id**: An identifier representing the current scan. This value will typically be incremented with each invocation.

**S3 Bucket Creation**:

The function first checks if an S3 bucket exists for storing CSV and PDF reports. The bucket name is constructed based on the environment using **os.getenv('environment')**. If the bucket doesn't exist, it is created in the **us-east-2** AWS region.

**Font and Default Parameters**:

The function sets up a default font and a dictionary of default parameters for different table names. These parameters include the columns that should be included in the report.

**Table Loop**:

The function iterates through each table name in the **table\_names\_list**.

**SQL Query Execution**:

For each table, it constructs an SQL query that selects data from that table for the given **scan\_id**. It then uses **pgs.run\_query\_using\_secrets** to execute the query using the provided secrets client and reporting database secret ARN. If an exception occurs during query execution, an empty result list is assigned.

**CSV and PDF Report Generation**:

The function prepares to generate CSV and PDF reports based on the retrieved data. It calculates the header list for the CSV file and the column sizes for formatting the PDF. It constructs file names for both the CSV and PDF reports based on the table name and scan ID.

If the CSV file does not exist, it is created and populated with data. If the CSV file already exists, data is appended to it. The function calculates the total width and maximum column height needed for PDF generation based on the column sizes. It converts the CSV file to a PDF file using a custom **convert** function. The PDF is then saved locally.

**File Upload to S3**:

The function uploads both the CSV and PDF files to the S3 bucket under appropriate folders (**csv\_files/** and **pdf\_files/**) using the S3 client.

# CIS Statements Used & Purpose

Report 1:

================

-- report name : database\_permissions

-- provide a summary of database privileges for each role in the current PostgreSQL database.

-- It lists the roles, the database they have privileges on, the types of privileges they have (CONNECT, CREATE, TEMPORARY), and whether they can log in.

SELECT

r.rolname,

datname,

ARRAY\_AGG(

CASE

WHEN has\_database\_privilege(r.rolname, c.oid, 'CONNECT') THEN 'CONNECT'

WHEN has\_database\_privilege(r.rolname, c.oid, 'CREATE') THEN 'CREATE'

WHEN has\_database\_privilege(r.rolname, c.oid, 'TEMPORARY') THEN 'TEMPORARY'

WHEN has\_database\_privilege(r.rolname, c.oid, 'TEMP') THEN 'CONNECT'

ELSE NULL

END

) AS privileges,

'DATABASE' AS level,

r.rolcanlogin

FROM

pg\_database c

JOIN

pg\_roles r ON has\_database\_privilege(r.rolname, c.oid, 'CONNECT,CREATE,TEMPORARY,TEMP')

WHERE

datname = current\_database()

GROUP BY

r.rolname, datname, r.rolcanlogin;

================

Report 2:

================

-- report name : schema\_privs\_role

-- summary of schema privileges for each role and schema combination

-- showing which roles have 'CREATE' and 'USAGE' privileges on each schema in the current PostgreSQL database.

SELECT

r.rolname,

current\_database() AS catalog\_name,

n.nspname AS schema\_name,

'SCHEMA' AS level,

'DATABASE' AS database\_name,

(

SELECT ARRAY\_AGG(priv)

FROM (

SELECT

CASE WHEN has\_schema\_privilege(r.rolname, n.nspname, 'CREATE') THEN 'CREATE' END AS priv

UNION

SELECT

CASE WHEN has\_schema\_privilege(r.rolname, n.nspname, 'USAGE') THEN 'USAGE' END AS priv

) AS privs

WHERE priv IS NOT NULL

) AS privs,

r.rolcanlogin

FROM

pg\_namespace n

JOIN

pg\_roles r ON true

WHERE

has\_schema\_privilege(r.rolname, n.nspname, 'CREATE,USAGE')

AND n.nspname NOT LIKE 'pg\_temp%'

AND n.nspowner <> r.oid;

================

Report 3:

================

-- report name : role\_specific\_privs

-- identify and list database roles (owners) that have specific privileges (CREATE and USAGE) on schemas in the current database.

-- includes information about the database, schema, and whether the role can log in.

SELECT

r.rolname,

current\_database() AS catalog\_name,

c.schema\_name,

'SCHEMA' AS level,

'DATABASE' AS database\_name,

'SCHEMA OWNER' AS privilege,

r.rolcanlogin

FROM

information\_schema.schemata c

JOIN

pg\_roles r ON c.schema\_owner = r.rolname

WHERE

has\_schema\_privilege(r.rolname, c.schema\_name, 'CREATE,USAGE')

AND c.schema\_name NOT LIKE 'pg\_temp%';

================

Report 4:

================

-- report name : role\_priv\_tables

-- identify and list the owners of tables in the database, along with their privileges.

-- It checks whether a role has the privilege to use tables in certain schemas and excludes system schemas from consideration.

-- includes information about the database, table, and whether the role can log in.

SELECT

r.rolname,

current\_database() AS catalog\_name,

'DATABASE' AS database\_name,

c.oid::regclass AS table\_name,

'TABLE' AS level,

'TABLE OWNER' AS privilege,

r.rolcanlogin

FROM

pg\_class c

JOIN

pg\_roles r ON c.relowner = r.oid

JOIN

pg\_namespace n ON c.relnamespace = n.oid

WHERE

n.nspname NOT IN ('information\_schema', 'pg\_catalog', 'sys')

AND c.relkind = 'r'

AND has\_schema\_privilege(r.rolname, n.oid, 'USAGE');

================

Report 5:

================

-- report name : role\_specific\_privs\_table

-- identify and list the roles that have specific privileges on tables in the database.

-- It checks for a set of privileges on tables in non-system schemas while excluding cases where the role owns the table.

-- includes information about the database, table, and whether the role can log in.

SELECT

r.rolname,

current\_database() AS catalog\_name,

'DATABASE' AS database\_name,

c.oid::regclass AS table\_name,

'TABLE' AS level,

ARRAY(

SELECT privs

FROM unnest(ARRAY[

(CASE WHEN has\_table\_privilege(r.rolname, c.oid, 'SELECT') THEN 'SELECT' ELSE NULL END),

(CASE WHEN has\_table\_privilege(r.rolname, c.oid, 'INSERT') THEN 'INSERT' ELSE NULL END),

(CASE WHEN has\_table\_privilege(r.rolname, c.oid, 'UPDATE') THEN 'UPDATE' ELSE NULL END),

(CASE WHEN has\_table\_privilege(r.rolname, c.oid, 'DELETE') THEN 'DELETE' ELSE NULL END),

(CASE WHEN has\_table\_privilege(r.rolname, c.oid, 'TRUNCATE') THEN 'TRUNCATE' ELSE NULL END),

(CASE WHEN has\_table\_privilege(r.rolname, c.oid, 'REFERENCES') THEN 'REFERENCES' ELSE NULL END),

(CASE WHEN has\_table\_privilege(r.rolname, c.oid, 'TRIGGER') THEN 'TRIGGER' ELSE NULL END)

]) AS privs

WHERE privs IS NOT NULL

) AS privileges,

r.rolcanlogin

FROM

pg\_class c

JOIN

pg\_roles r ON c.relowner = r.oid

JOIN

pg\_namespace n ON c.relnamespace = n.oid

WHERE

n.nspname NOT IN ('information\_schema', 'pg\_catalog', 'sys')

AND c.relkind = 'r'

AND has\_table\_privilege(r.rolname, c.oid, 'SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER')

AND has\_schema\_privilege(r.rolname, c.relnamespace, 'USAGE')

AND c.relowner <> r.oid;

================

Report 6:

================

-- report name : views\_ownership\_usage\_privs

-- identify and list the roles that have the privilege to use (or access) views in the database.

-- checks for schema usage privilege on views in non-system schemas while excluding system schemas and includes information about the database, view, and whether the role can log in.

-- The query specifically focuses on view ownership and usage privileges.

SELECT

r.rolname,

current\_database() AS current\_db,

'DATABASE' AS object\_type,

c.oid::regclass AS object\_name,

'VIEW' AS object\_kind,

'VIEW OWNER' AS object\_owner,

r.rolcanlogin

FROM

pg\_class c

JOIN

pg\_namespace n ON c.relnamespace = n.oid

JOIN

pg\_roles r ON c.relowner = r.oid

WHERE

n.nspname NOT IN ('information\_schema', 'pg\_catalog', 'sys')

AND c.relkind = 'v'

AND has\_schema\_privilege(r.rolname, c.relnamespace, 'USAGE');

================

Report 7:

================

-- report name : view\_privs\_role

-- identify and list the roles that have specific privileges on views in the database.

-- checks for schema usage privilege on views in non-system schemas while excluding system schemas and includes information about the database, view, and whether the role can log in.

-- constructs an array of view privileges for each role.

SELECT

r.rolname,

current\_database() AS current\_db,

'DATABASE' AS object\_type,

c.oid::regclass AS object\_name,

'VIEW' AS object\_kind,

'VIEW OWNER' AS object\_owner,

ARRAY(

SELECT privs

FROM unnest(ARRAY['SELECT', 'INSERT', 'UPDATE', 'DELETE', 'TRUNCATE', 'REFERENCES', 'TRIGGER']) AS privs

WHERE

(privs = 'SELECT' AND has\_table\_privilege(r.rolname, c.oid, 'SELECT')) OR

(privs = 'INSERT' AND has\_table\_privilege(r.rolname, c.oid, 'INSERT')) OR

(privs = 'UPDATE' AND has\_table\_privilege(r.rolname, c.oid, 'UPDATE')) OR

(privs = 'DELETE' AND has\_table\_privilege(r.rolname, c.oid, 'DELETE')) OR

(privs = 'TRUNCATE' AND has\_table\_privilege(r.rolname, c.oid, 'TRUNCATE')) OR

(privs = 'REFERENCES' AND has\_table\_privilege(r.rolname, c.oid, 'REFERENCES')) OR

(privs = 'TRIGGER' AND has\_table\_privilege(r.rolname, c.oid, 'TRIGGER'))

) AS object\_privileges,

r.rolcanlogin

FROM

pg\_class c

JOIN pg\_namespace n ON c.relnamespace = n.oid

JOIN pg\_roles r ON c.relowner = r.oid

WHERE

n.nspname NOT IN ('information\_schema', 'pg\_catalog', 'sys')

AND c.relkind = 'v'

AND has\_schema\_privilege(r.rolname, c.relnamespace, 'USAGE')

AND c.relowner <> r.oid;

================

Report 8:

================

-- report name : sequence\_ownership\_usage\_privs

-- identify and list the roles that have specific privileges (SELECT and UPDATE) on sequences in the database.

-- checks for schema usage privilege on sequences in non-system schemas while excluding system schemas and includes information about the database, sequence, and whether the role can log in.

-- focuses on sequence ownership and usage privileges.

SELECT

r.rolname,

current\_database() AS current\_db,

'DATABASE' AS object\_type,

c.oid::regclass AS object\_name,

'SEQUENCE' AS object\_kind,

'SEQUENCE OWNER' AS object\_owner,

r.rolcanlogin

FROM

pg\_class c

JOIN pg\_namespace n ON c.relnamespace = n.oid

JOIN pg\_roles r ON c.relowner = r.oid

WHERE

n.nspname NOT IN ('information\_schema', 'pg\_catalog', 'sys')

AND c.relkind = 'S'

AND has\_table\_privilege(r.rolname, c.oid, 'SELECT, UPDATE')

AND has\_schema\_privilege(r.rolname, c.relnamespace, 'USAGE')

AND c.relowner = r.oid;

================

Report 9:

================

-- report name : roles\_specific\_privileges\_sequences

-- identify and list the roles that have both SELECT and UPDATE privileges on sequences in the database.

-- It filters out sequences in specific schemas, focuses on sequences, and includes information about the database, sequence, privileges, and whether the role can log in.

-- includes roles with both SELECT and UPDATE privileges on sequences.

WITH Privileges AS (

SELECT

r.rolname AS role\_name,

current\_database() AS database\_name,

'DATABASE' AS object\_type,

c.oid::regclass AS object\_name,

'SEQUENCE' AS privilege\_type,

CASE WHEN has\_table\_privilege(r.rolname, c.oid, 'SELECT') THEN 'SELECT' ELSE NULL END AS select\_priv,

CASE WHEN has\_table\_privilege(r.rolname, c.oid, 'UPDATE') THEN 'UPDATE' ELSE NULL END AS update\_priv,

r.rolcanlogin AS can\_login

FROM

pg\_class c

JOIN

pg\_namespace n ON c.relnamespace = n.oid

JOIN

pg\_roles r ON r.oid = c.relowner

WHERE

n.nspname NOT IN ('information\_schema', 'pg\_catalog', 'sys')

AND c.relkind = 'S'

AND has\_table\_privilege(r.rolname, c.oid, 'SELECT,UPDATE')

AND has\_schema\_privilege(r.rolname, c.relnamespace, 'USAGE')

)

SELECT

role\_name,

database\_name,

object\_type,

object\_name,

privilege\_type,

ARRAY\_REMOVE(ARRAY[select\_priv, update\_priv], NULL) AS privileges,

can\_login

FROM

Privileges

WHERE

ARRAY[select\_priv, update\_priv] IS NOT NULL;

================

Report 10:

================

-- report name : roles\_privs\_fdw

-- identify and list the roles that have the USAGE privilege on foreign data wrappers (FDWs) in the database.

-- focuses on FDW ownership and USAGE privileges.

-- which roles have the authority to use FDWs in the database.

SELECT

r.rolname AS role\_name,

current\_database() AS database\_name,

'DATABASE' AS object\_type,

fdwname AS object\_name,

'FDW' AS privilege\_type,

'FDW OWNER' AS privilege\_name,

r.rolcanlogin AS can\_login

FROM

pg\_catalog.pg\_foreign\_data\_wrapper fdw

JOIN

pg\_catalog.pg\_roles r ON fdw.fdwowner = r.oid

WHERE

has\_foreign\_data\_wrapper\_privilege(r.rolname, fdwname, 'USAGE');

================

Report 11:

================

-- report name : roles\_login\_fdw

-- identify and list the roles that have the USAGE privilege on foreign data wrappers (FDWs) in the database.

-- constructs an array of privileges for each role and excludes cases where the role owns the FDW.

-- which roles can use FDWs in the database and whether they can log in

SELECT

r.rolname AS role\_name,

current\_database() AS database\_name,

'DATABASE' AS object\_type,

fdwname AS object\_name,

'FDW' AS privilege\_type,

ARRAY[CASE WHEN has\_foreign\_data\_wrapper\_privilege(r.rolname, fdwname, 'USAGE') THEN 'USAGE' ELSE NULL END] AS privileges,

r.rolcanlogin AS can\_login

FROM

pg\_catalog.pg\_foreign\_data\_wrapper

JOIN

pg\_catalog.pg\_roles r ON fdwowner = r.oid

WHERE

has\_foreign\_data\_wrapper\_privilege(r.rolname, fdwname, 'USAGE')

AND fdwowner <> r.oid;

================

Report 12:

================

-- file name : roles\_privs\_language

-- identify and list the roles that have the USAGE privilege on languages in the database.

-- constructs an array of privileges for each role based on the USAGE privilege on specific languages.

-- understand which roles can use particular languages for writing stored procedures and functions in the database and whether they can log in.

SELECT

r.rolname AS role\_name,

current\_database() AS database\_name,

'DATABASE' AS object\_type,

l.lanname AS object\_name,

'LANGUAGE' AS privilege\_type,

ARRAY[CASE WHEN has\_language\_privilege(r.rolname, l.lanname, 'USAGE') THEN 'USAGE' ELSE NULL END] AS privileges,

r.rolcanlogin AS can\_login

FROM

pg\_catalog.pg\_language l

JOIN

pg\_catalog.pg\_roles r ON has\_language\_privilege(r.rolname, l.lanname, 'USAGE');

================================

Report 13:

================================

-- report name : functions\_ownership\_roles

-- identify and list the database roles that own functions in the database

-- provides information about the roles, the functions they own, and whether the roles can log in

-- useful for understanding the ownership of functions within the database.

SELECT

r.rolname AS role\_name,

current\_database() AS database\_name,

'DATABASE' AS object\_type,

n.nspname || '.' || p.proname AS object\_name,

'FUNCTION' AS privilege\_type,

'FUNCTION OWNER' AS privilege\_name,

r.rolcanlogin AS can\_login

FROM

pg\_proc p

JOIN

pg\_namespace n ON p.pronamespace = n.oid

JOIN

pg\_roles r ON r.oid = p.proowner

WHERE

n.nspname <> 'pg\_catalog';

==============================================================================

Report 14:

==============================================================================

-- report name : user\_roles

-- Purpose of this SQL is to retreive information about database roles and their associated permissions

-- It reports roles that meet specific criterion related to their permissions and roles, including superuser status, replication permissions

-- role creation permissions and database creation permissions.

select a.oid as user\_role\_id

, a.rolname as user\_role\_name

, a.rolcanlogin as role\_can\_login

, b.roleid as other\_role\_id

, c.rolname as other\_role\_name

, a.rolreplication as has\_replication\_perm

, a.rolcreaterole as has\_createrole\_perm

, a.rolcreatedb as has\_createdb\_perm

from pg\_roles a

inner join pg\_auth\_members b on a.oid=b.member

inner join pg\_roles c on b.roleid=c.oid

WHERE

a.rolsuper = true

OR a.rolreplication = true

OR a.rolcreaterole = true

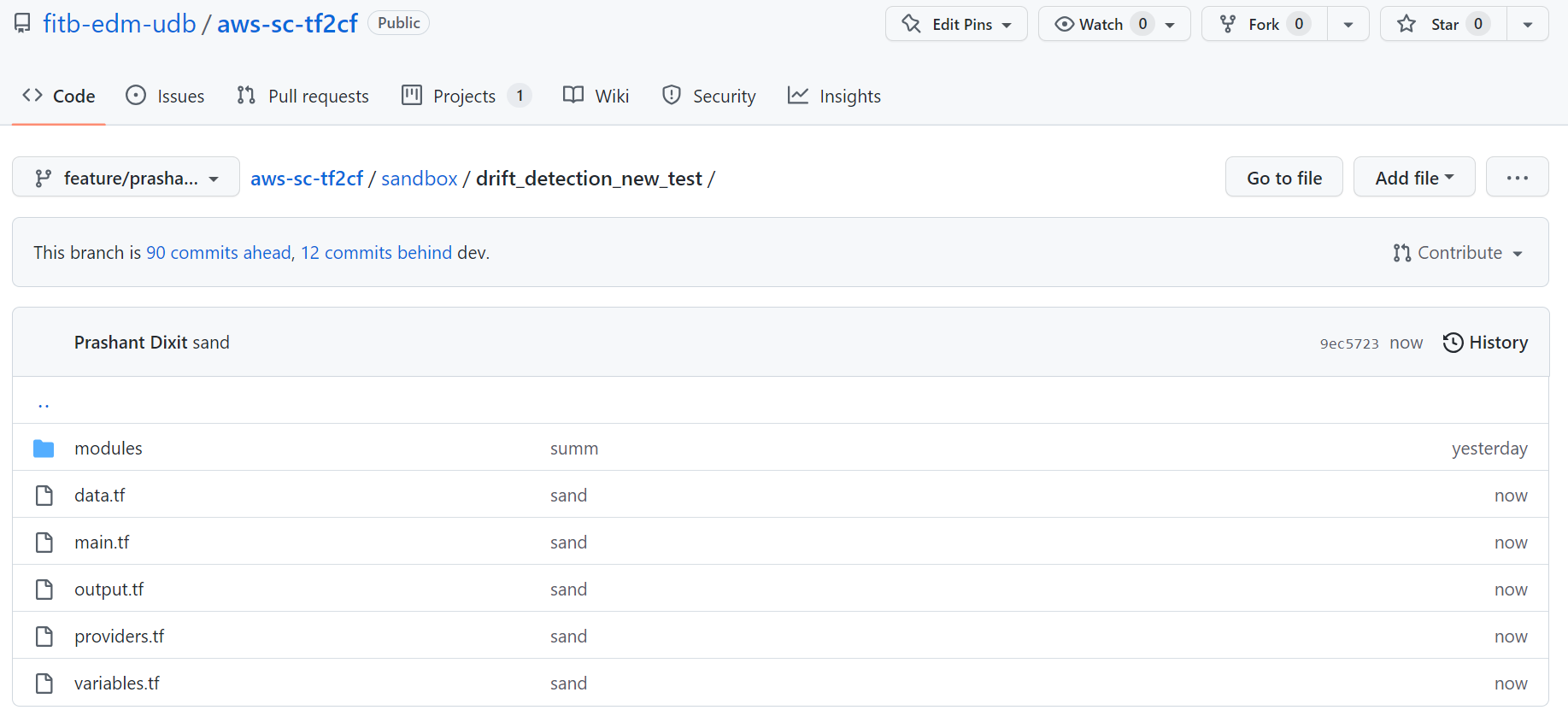
OR a.rolcreatedb = true;

# How to call the function?

The code should be deployed via Terraform Cloud for which you have to create respective code repository in GitHub and integrate it with the Terraform to deploy the lambda from the GUI.

1. Steps to Integrate GitHub and Terraform Cloud:

First step is to create a feature edition out of the master branch for environment, for example, I have created a new folder for ‘sandbox’ environment (see image below) where I have created all the terraform files that will help to deploy the code in respective environment.

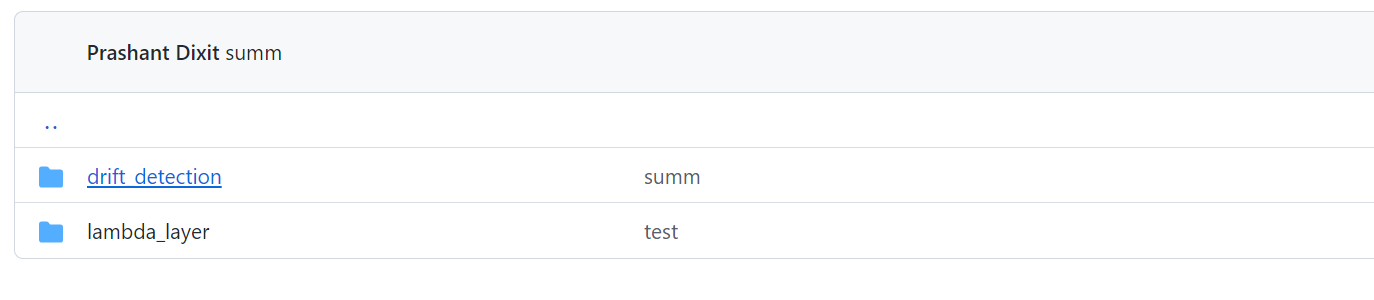


1. Create folders for Lambda and Layers

Create a separate folder to place the drift detection code and lambda layers (csv2pdf, postgres\_utils) which you will call via terraform code (main.tf)

For better understanding, please review below directory hierarchy for sandbox.

Link 🡪 <https://github.info53.com/fitb-edm-udb/aws-sc-tf2cf/tree/feature/prashant/sandbox/drift_detection_new_test>



1. Update environment details into the main config file

Next is you update environment specific details in the main config file.

For example, sandbox’s main config file is present under : <https://github.info53.com/fitb-edm-udb/aws-sc-tf2cf/blob/feature/prashant/sandbox/drift_detection_new_test/main.tf>

This is where we define resource information, their names and attributes i.e. Security Groups, Subnet IDs, Lambda name, policy, role name, timeouts values (15 mins def), layers information, lambda function map (source code location, handler info, account names, secret pattern, secret ARN etc.)

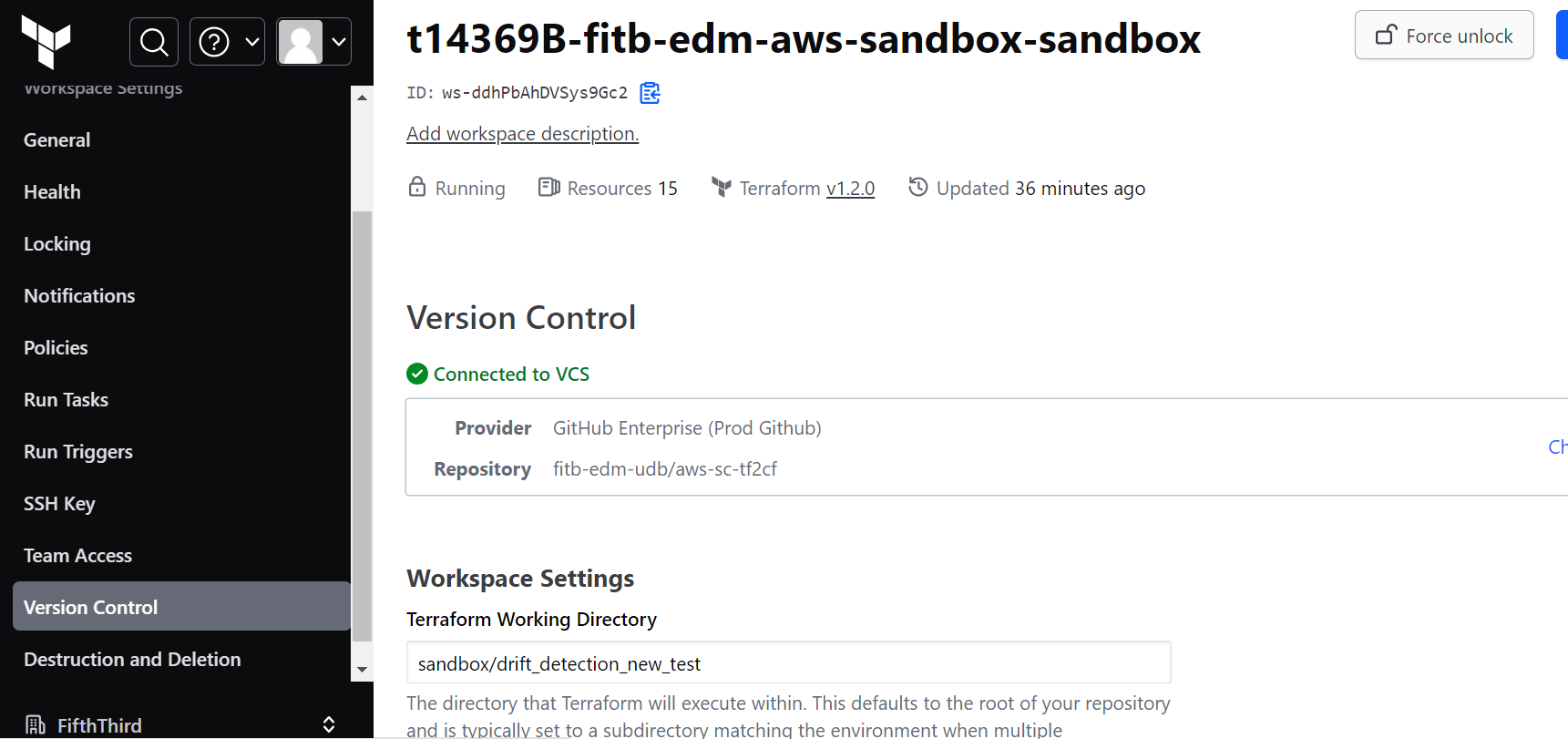
1. Prepare a SOURCE CODE repository.

This will be the location where you put core source codes, for example (sandbox repo : <https://github.info53.com/fitb-edm-dba/adithya_lambdas/tree/feature/create_lambda>)

This is where you create core config file (second main.tf) where you defined resources information and their parameters i.e., cloudwatch log groups, AssumeRole information, IAM Role policies, RDS Policies, S3 Policies and rules, Inbound rules, Secret Manager polices, VPC policies etc.

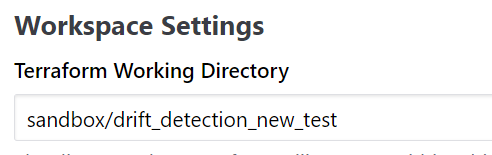
1. Map Terraform codes with repos.

Next is you map your GitHub repositories, your code base with the Terraform workspace. As a standard method, everything in bank we deploy it using Terraform and that’s how it automatically manages the state files for every change we do for the objects.

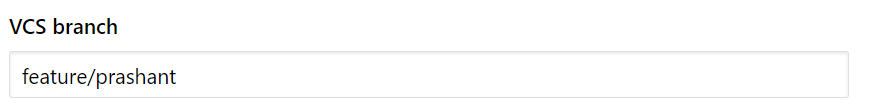


Two important workspace settings are:

* The working directory where you have kept the code in GitHub.

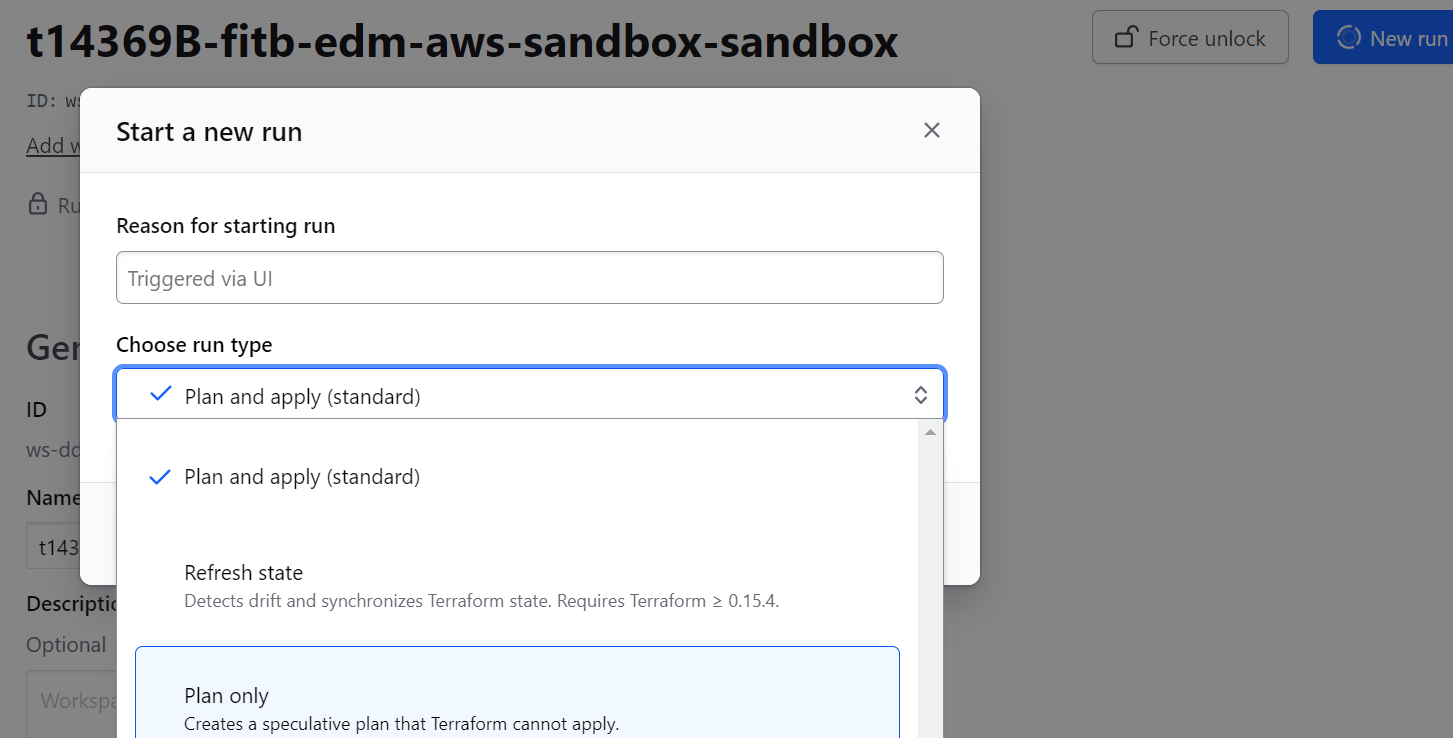


* VCS Branch or the Version Control Branch:



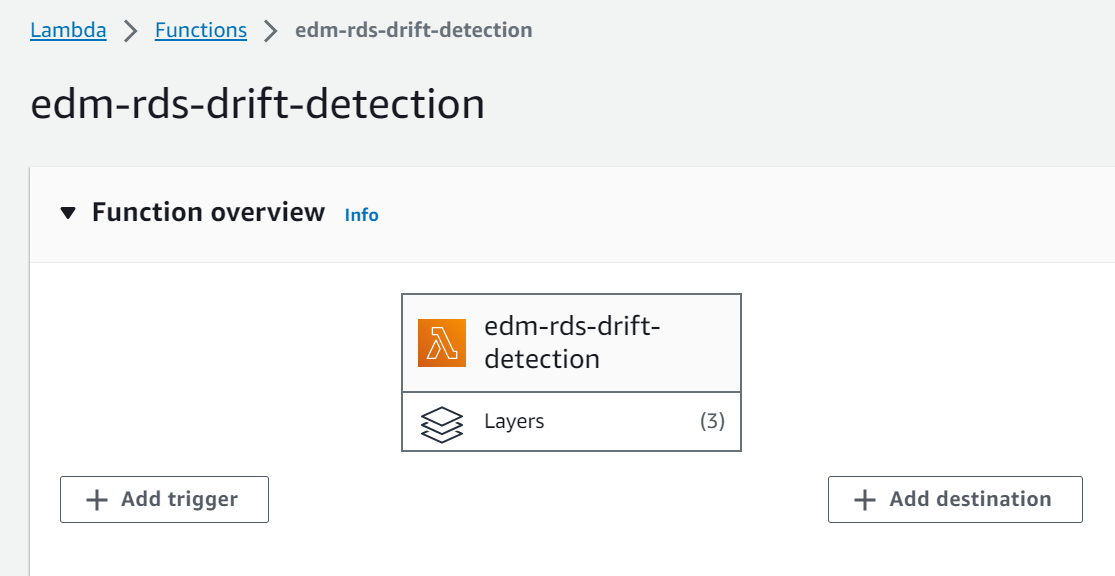
1. Run the code from Terraform Console.

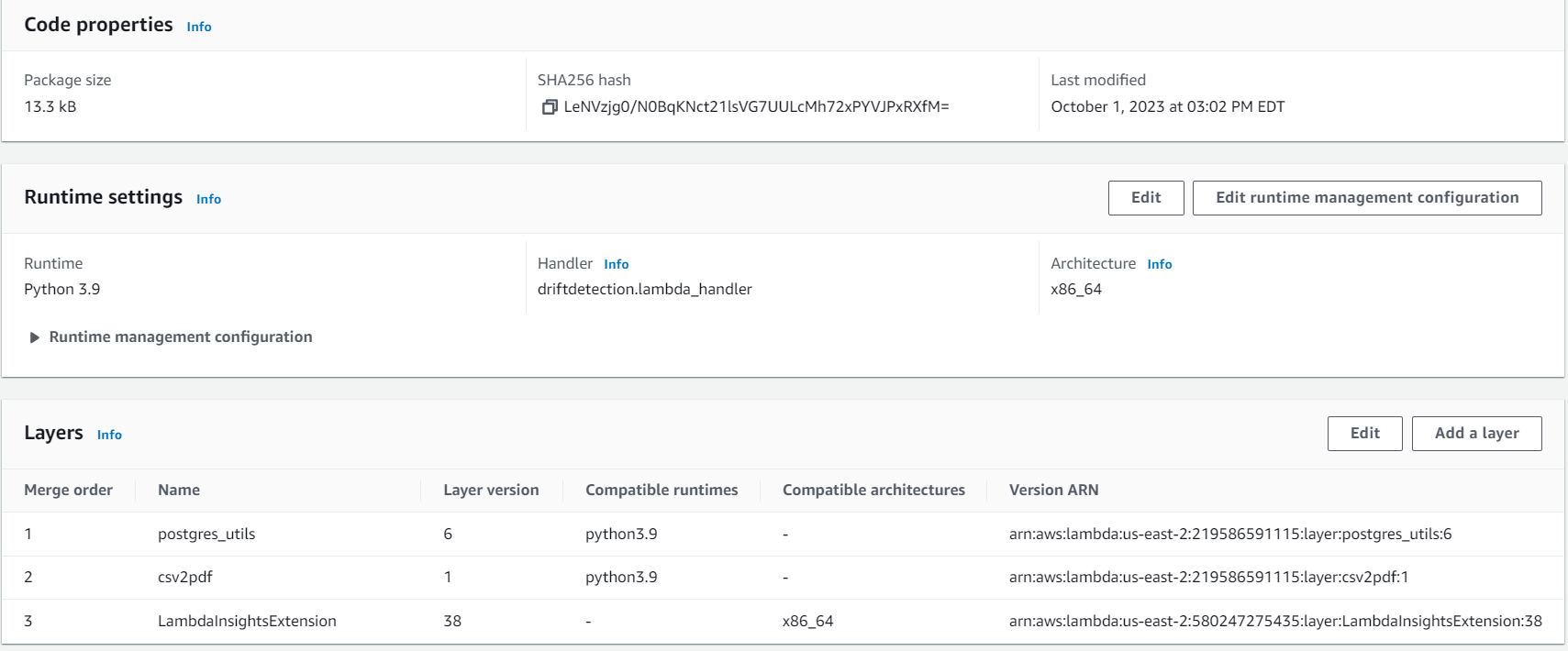
Once done with all mappings between TF and GitHub, next is you run the code. It’s always recommended to run PLAN to see what all changes the code is going to do (delete, add, modify etc.). Once all good, only then you should run the APPLY command and that will make all changes persistent.



1. Verify if the Function is created.

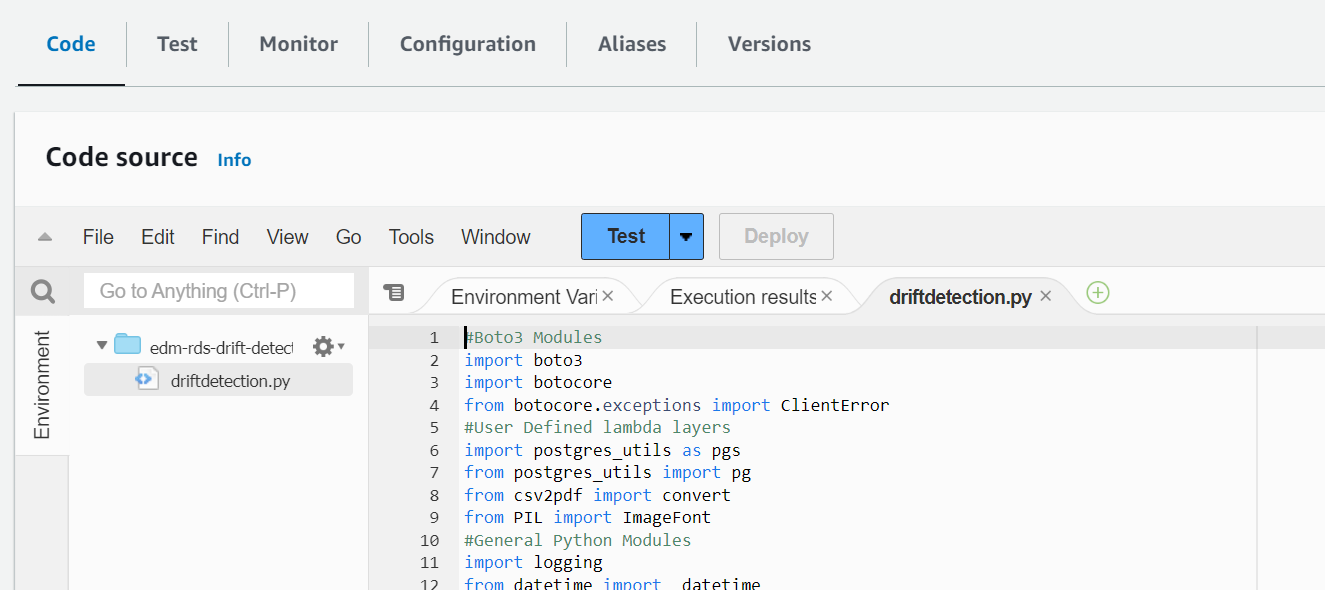
Login into respective AWS environment to verify if function is created with all privileges, rules and permissions.

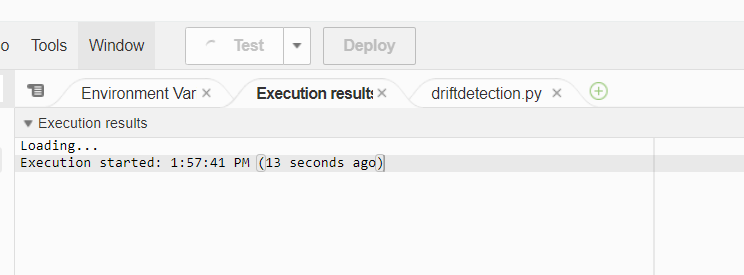




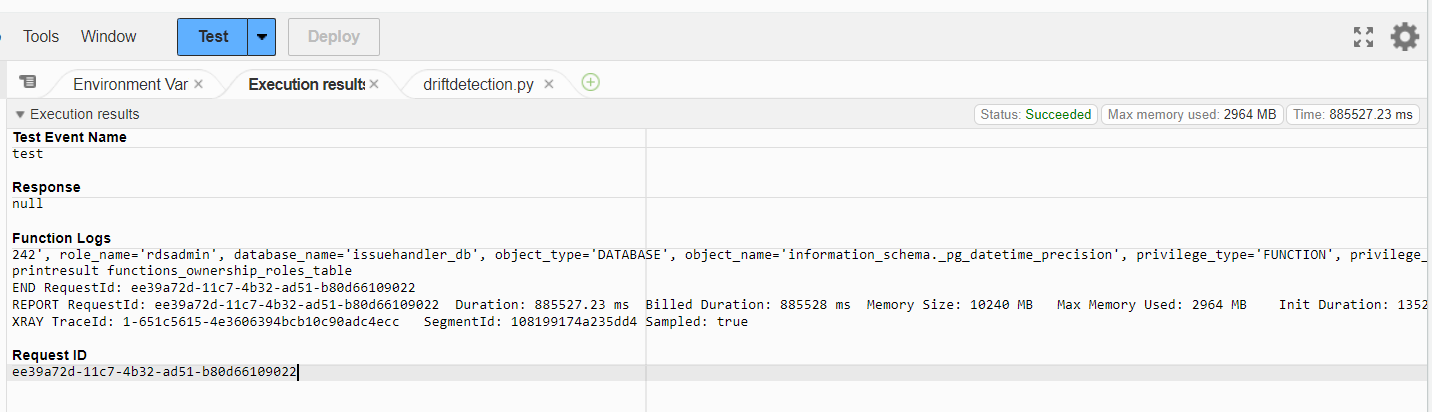
1. Run the function manually to generate reports.

Once the function is created, next you can execute it to see if its working. Click on the button ‘Test’ and that will call the embedded code.



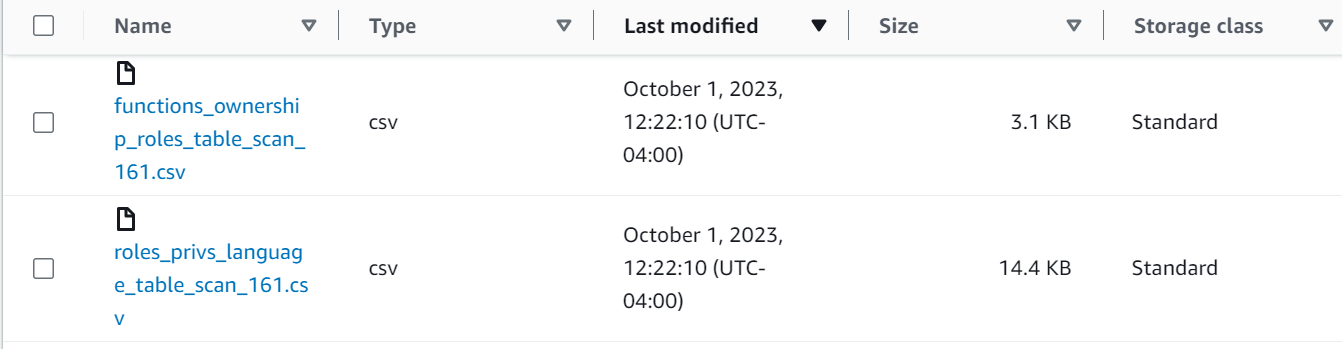


Once completed, you will either see error message, or a completion message with elapsed time, memory consumption and other details.



1. Verify reports under S3 bucket.

Once the function execution is completed, you will see CSV & PDF reports are getting generated in the S3 bucket under environment name you have used in the Terraform main config file.



# Monitoring & Debugging

The best way to monitor the Lambda is by adding debugging/logging (print) lines into the Python code to understand which part of line of the code is breaking. The other AWS specific monitoring methods are cloud watch logs.

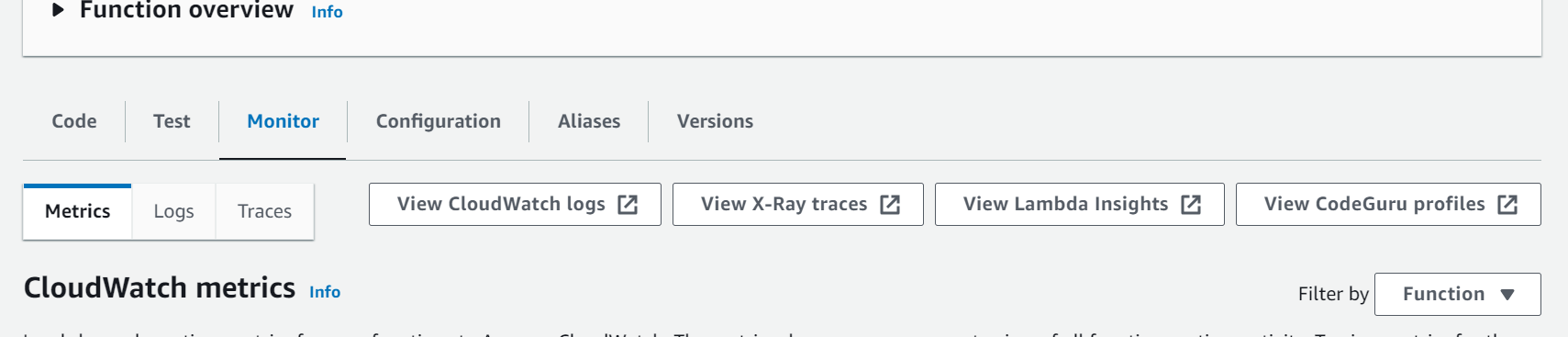
AWS Lambda automatically monitors Lambda functions and sends log entries to Amazon CloudWatch. It comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation and other output from your function's code to the log stream.

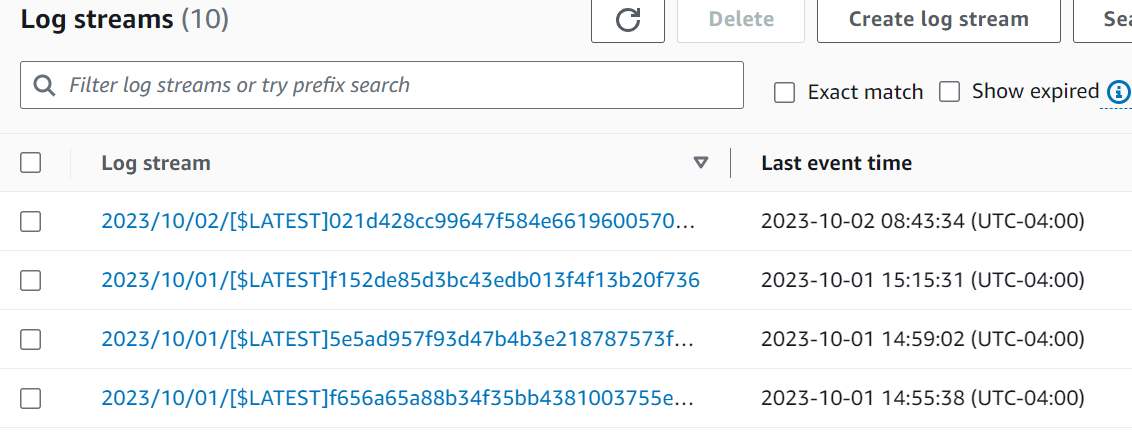
Though the code also uses logging module (logging) in the standard library which is by default set to INFO level which you can change anytime to get more runtime insights.

import logging

logger = logging.getLogger()

logger.setLevel(logging.INFO)





# Add new reports to the Code.

The lambda function has ~ 20 different custom SQLs which adheres to the CIS standard and helps to identify the drift in all RDS instances in local and cross accounts. Below are the ways how you can add more reports into existing code.

For example: If you would like to add a new SQL statement to identify and list the roles that have the privilege to use (or access) views in the database. Something that checks for schema usage privilege on views in non-system schemas while excluding system schemas and includes information about the database, view, and whether the role can log in.

* You have prepared the statement that you would like to use.

SELECT

r.rolname,

current\_database() AS current\_db,

'DATABASE' AS object\_type,

c.oid::regclass AS object\_name,

'VIEW' AS object\_kind,

'VIEW OWNER' AS object\_owner,

r.rolcanlogin

FROM

pg\_class c

JOIN

pg\_namespace n ON c.relnamespace = n.oid

JOIN

pg\_roles r ON c.relowner = r.oid

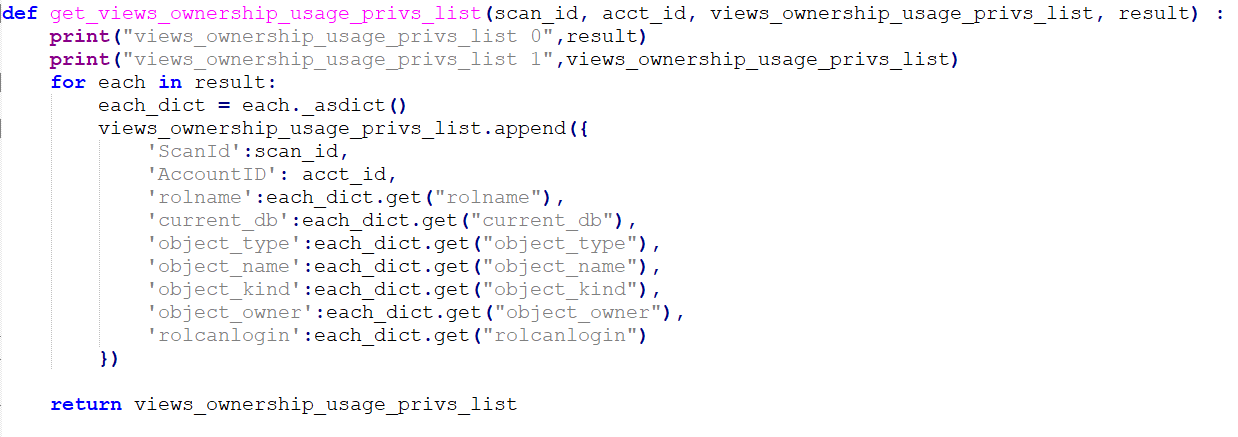
WHERE

n.nspname NOT IN ('information\_schema', 'pg\_catalog', 'sys')

AND c.relkind = 'v'

AND has\_schema\_privilege(r.rolname, c.relnamespace, 'USAGE');

* Next is you write a function before the ‘lambda handler’.



The purpose of the Python function get\_views\_ownership\_usage\_privs\_list is to take certain input data (scan\_id, acct\_id, result), and combine this data with an existing list (views\_ownership\_usage\_privs\_list) to create a list of dictionaries.

It takes four parameters as input:

scan\_id: This is an identifier for a scanning operation.

acct\_id: This is an account identifier.

views\_ownership\_usage\_privs\_list: This is an existing list that contains dictionaries.

result: This is a collection of data, a list of named tuples.

The function starts by printing the initial state of the result and views\_ownership\_usage\_privs\_list lists. It then iterates over each element in the result collection. Each element appears to be a named tuple because it is converted to a dictionary using \_asdict(). For each element in result, it extracts specific fields from the named tuple (e.g., 'rolname', 'current\_db', 'object\_type', etc.) and creates a dictionary containing these fields along with scan\_id and acct\_id.

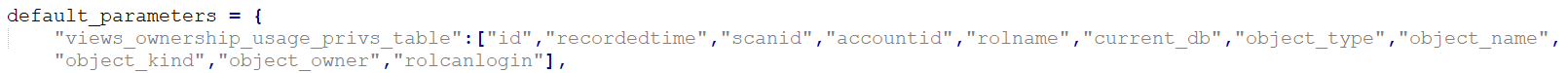
This dictionary is then appended to the views\_ownership\_usage\_privs\_list list. After processing all elements in result, the function returns the updated views\_ownership\_usage\_privs\_list, which now contains additional dictionaries representing the properties of the items in result.

In summary, this function will take the data from result, processes it, and adds the processed data to an existing list of dictionaries, which is then returned as the result.

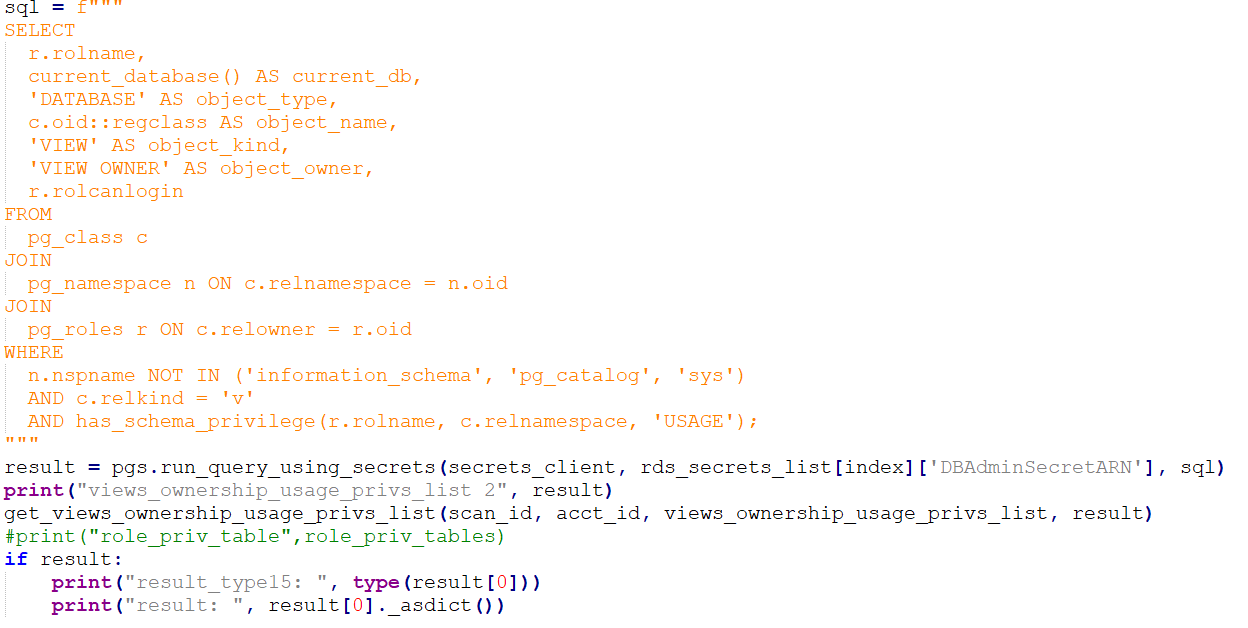
* Next step is to create an empty list where the result will be placed.



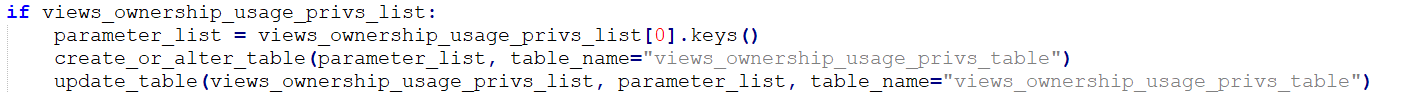
* Next step is to add it the generate\_csv\_and\_pdf\_reports\_for\_the\_drift\_tables function where I have created a parameterized list called “default\_parameters” which helps to create an empty report in case when the SQL query results in zero.



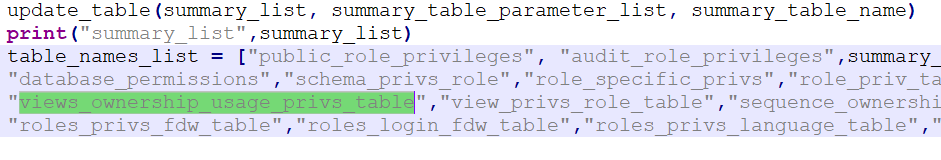
* Next is to add the SQL to the handler and this will execute the statement to retrieve the information from the PostgreSQL database, filtering the results based on defined conditions, and then processing and appending the results to a list using the ‘get\_views\_ownership\_usage\_privs\_list’ function.



* Next step is to add the list to the if-else condition and create a new table with some name and with update conditions to append data into the table.



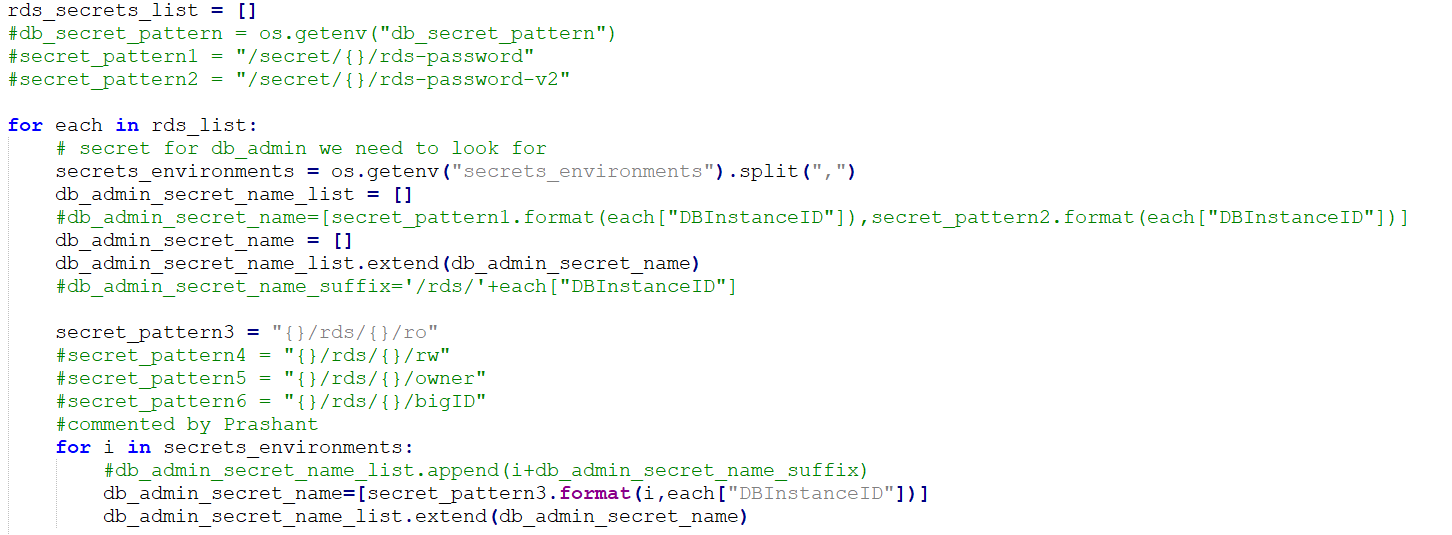
* Finally, you must update the summary table.



# Lambda Function 2 : One to cover only new patterns ro/rw/BigID/Owner formats.

This section explains about the second lambda function that we have created to cover additional search patterns – ro/rw/bigid and owner. We have tried to add it into the main lambda function (function 1) but its lead to an additional load on it and caused it to fail as the execution failed to get complete within limit of 900 seconds. Hence, decided to write a new one with only a specific pattern should be enabled at once.

Below are in depth details about the section that is different than the original (lambda 1). Overall, this code is a part of a process for generating a list of secret names based on patterns specific to different RDS instances and environments.



Let's break down what this code is doing:

- *Initialize Variables:*

* rds\_secrets\_list: This is an empty list that will be used to store the generated secret names.
* secrets\_environments: This variable is used to contain a list of environments or prefixes for secrets.

*- Loop Through RDS Instances:*

The code iterates through a list of rds\_list, which contain information about various RDS instances. It's a list of dictionaries where each dictionary represents an RDS instance.

- *Generate Secret Names:*

Inside the loop, there is a section of code that generates secret names for different purposes (e.g., read-only, read-write, owner, etc.) based on specific patterns. These patterns depend on the environment and the ID of each RDS instance. The secret names are stored in the db\_admin\_secret\_name\_list.

In the code snippet, **secret\_pattern3** is used to generate a secret name for read-only access. There are commented out patterns for other types of access (secret\_pattern4, secret\_pattern5, secret\_pattern6), but they are not used in this loop.

*- List Extension:*

The generated secret names are extended (appended) to the db\_admin\_secret\_name\_list using the extend method. This means that, for each RDS instance and environment, a list of corresponding secret names is built up.

- *Iteration Through Environments:*

The loop also iterates through different environments, and for each environment, it appends a secret name suffix specific to that environment.