**INTRODUCTION**

**Project Goal** The objective of this project is to automate the process of transferring compressed files (.gz) from an FTP server to an Amazon S3 bucket.

**AWS Lambda Function**

AWS Lambda is a serverless compute service that enables you to run code without the need to provision or manage any servers. With Lambda, code can be executed in response to various triggers, such as data changes in an Amazon S3 bucket or at scheduled intervals through Amazon EventBridge (formerly CloudWatch Events).

In our project, the Lambda function is scheduled to run daily at 06:00 AM to transfer new files from the FTP server to the S3 bucket. This schedule is managed via Amazon EventBridge. By leveraging Lambda’s serverless architecture, we eliminate the need to handle server maintenance, scaling, or capacity planning, as these are automatically managed by AWS.

The function is written in Python, a language natively supported by AWS Lambda, allowing for straightforward development and integration. Additionally, Lambda operates on a Pay-As-You-Go pricing model, where costs are based on the number of function requests and the compute time used (measured in milliseconds). This pricing approach can be more economical than maintaining a dedicated server continuously.

The primary purpose of this AWS Lambda function is to automate the retrieval and processing of new files from the FTP server and store them directly in the S3 bucket. This includes both the extraction and transfer of data. AWS Lambda’s flexibility and ease of integration make it well-suited for this task, especially since it can also handle real-time data processing when needed.

**AWS S3 Bucket**

Amazon S3 (Simple Storage Service) is a scalable, secure, and highly durable object storage service provided by AWS. An S3 bucket is a container within Amazon S3 where you can store, organize, and manage data objects such as files, images, videos, backups, and other data types. S3 buckets are globally accessible, but they can be configured with region-specific settings to control data locality and latency.

S3 buckets can store an unlimited amount of data, making them suitable for a wide range of storage needs, from small backups to large data lakes.

**FTP Server**

An FTP (File Transfer Protocol) server is a computer system or software that uses the File Transfer Protocol to store, transfer, and manage files over a network, typically the internet. FTP servers are commonly used for sharing and exchanging files between computers, especially when large files need to be transferred or multiple files need to be organized in a directory structure. FTP servers require users to authenticate with a username and password.

**PROJECT SCOPE**

This documentation provides a comprehensive overview of the project, beginning with the goals and objectives. It outlines the prerequisites necessary for successfully deploying the project.

The document includes a complete breakdown of the code used to deploy the AWS Lambda function, with a step-by-step explanation to facilitate understanding and replication. A section on monitoring and logging follows, offering insights into how AWS services can be used to track function performance and troubleshoot issues effectively.

The conclusion provides a summary of the project’s outcomes, as well as suggestions for future improvements or additional features that could be implemented. An appendix at the end of the documentation includes all relevant sources and a glossary, which defines key terms and concepts to support users in navigating the material.

**GOALS AND OBJECTIVES**

The primary goal of this project is to automate the replication of new .gz files from an FTP server to an AWS S3 bucket, enabling these files to be accessed on a scheduled basis within a Snowflake environment. To achieve this, an AWS Lambda function has been implemented to streamline and automate the file transfer process.

The project aims to create an efficient, cost-effective solution that optimizes both performance and ease of use. Given the large volume of files on the FTP server, it is also essential to incorporate error-handling mechanisms to prevent issues that could affect the processing and loading of data into Snowflake.

**PRE-REQUISITES**

To execute the AWS Lambda function, the primary requirement is an active AWS account. For extracting files from the FTP server, you will need the FTP server credentials, including the FTP host, username, and password.

Additionally, the Python code used for processing FTP files requires the **ftplib** library to interact with the FTP server and the AWS SDK **boto3** library to connect with AWS services.

**IMPLEMENTATION STEPS**

1. **Creating S3 Bucket**

**Go to the AWS Services, select S3 and create an S3 bucket with default settings.**

1. **Creating an IAM (Identity and Access Management)** **Policy**

Navigate to the IAM Console, then go to Policies and select "Create Policy." Choose the S3 service, and under "Actions Allowed," select "All List Actions," "All Read Actions," "All Permissions Management Actions," and "All Tagging Actions." Set the resources to "All Resources."

Next, add a permission for CloudWatch Logs, applying the same settings as for the S3 bucket. Finally, provide a name for the policy, review the services and permissions granted, and create the policy.

**Why we need to create an IAM Policy?**

An IAM policy is essential for an AWS Lambda function because it defines what actions the function is authorized to perform on other AWS services and resources. In AWS, IAM policies are used to manage permissions securely, and Lambda functions operate with the permissions that are granted to them through attached IAM policies.

By assigning this IAM Policy the Lambda function is granted the access to S3 Bucket and Cloudwatch for logging and thus it can perform action such as storing the files in S3 bucket

Attaching an IAM policy allows you to enforce the **least privilege principle**, which means granting only the minimum permissions necessary for the function to perform its task. This minimizes security risks by limiting what the Lambda function can do.

AWS automatically uses the IAM policy attached to the Lambda function's execution role to determine what resources and actions are permitted.

1. **Creating an IAM Role and attaching IAM Policy**:

Navigate to the IAM Console and select "Create Role." Choose the Lambda service for which this role will be utilized. In the permissions policies section, search for and select the policy you created earlier by name. Finally, assign a name to the role, review the overview, and create the role.

**Why an IAM Role is required?**

IAM roles allow you to grant specific permissions to Lambda functions without using long-term credentials. This enhances security by avoiding hard-coded access keys and secret keys within the function code.

By attaching a role to a Lambda function, you can enforce the least privilege principle, which means granting only the permissions necessary for the function to perform its tasks. This reduces the risk of unintended access to sensitive resources.

By creating an IAM policy and attaching it to your Lambda function's role, you ensure that the function has controlled, secure access to the AWS resources it needs to function properly. This approach follows AWS’s best practices for security and resource management

1. **Creating a Lambda function**

Navigate to AWS Services, then select **Compute** and choose **Lambda**.

* Select **Author from Scratch**, provide a name for the function, choose **Python** 3.12 as the language, and set the architecture to **x86\_64**.
* Under **Change Default Execution Role**, select **Use Existing Role** and choose the name of the role you created earlier. Finally, click **Create Function**.

1. **Coding in AWS Lambda Function**

In the code section of the Lambda Function, Paste the following code of FTP server

import json

import boto3

import ftplib

import botocore

from botocore.errorfactory import ClientError

from boto3.s3.transfer import TransferConfig

import os

from base64 import b64decode

import logging

import time

logger = logging.getLogger()

logger.setLevel(logging.INFO)

# encryption decryption for host

ENCRYPTED\_HOST = os.environ['FTP\_HOST']

DECRYPTED\_HOST = boto3.client('kms').decrypt(

    CiphertextBlob=b64decode(ENCRYPTED\_HOST),

    EncryptionContext={'LambdaFunctionName': os.environ['AWS\_LAMBDA\_FUNCTION\_NAME']}

)['Plaintext'].decode('utf-8')

# encryption decryption for  for user

ENCRYPTED\_USER = os.environ['FTP\_USER']

DECRYPTED\_USER = boto3.client('kms').decrypt(

    CiphertextBlob=b64decode(ENCRYPTED\_USER),

    EncryptionContext={'LambdaFunctionName': os.environ['AWS\_LAMBDA\_FUNCTION\_NAME']}

)['Plaintext'].decode('utf-8')

# encryption decryption for  for pswd

ENCRYPTED\_PSWD = os.environ['FTP\_PSWD']

DECRYPTED\_PSWD = boto3.client('kms').decrypt(

    CiphertextBlob=b64decode(ENCRYPTED\_PSWD),

    EncryptionContext={'LambdaFunctionName': os.environ['AWS\_LAMBDA\_FUNCTION\_NAME']}

)['Plaintext'].decode('utf-8')

FTP\_HOST = DECRYPTED\_HOST

FTP\_USER = DECRYPTED\_USER

FTP\_PSWD = DECRYPTED\_PSWD

#Bucket name

s3\_bucket\_name = <bucket\_name>

s3\_region\_name = <region\_name>

# Initialize S3 client

s3\_client = boto3.client(

   's3',

   aws\_access\_key\_id= <id>,

   aws\_secret\_access\_key= <key>,

   region\_name=s3\_region\_name

)

# checking if file exists in S3 bucket

def file\_exists\_in\_s3(s3\_client, bucket, key):

    try:

        s3\_client.head\_object(Bucket=bucket, Key=key)

        return True

    except ClientError as e:

        if e.response['Error']['Code'] == '404':

            return False

        else:

            raise

# Function to get the list of files in the S3 bucket

def list\_s3\_files(bucket\_name):

    s3\_files = []

    response = s3\_client.list\_objects\_v2(Bucket=bucket\_name)

    if 'Contents' in response:

        for obj in response['Contents']:

            s3\_files.append(obj['Key'])

    return s3\_files

# downloading file

def download\_file(ftp, ftp\_file, local\_filename, retries=3):

    """Download a file from the FTP server with retry mechanism."""

    for attempt in range(retries):

        try:

            with open(local\_filename, 'wb') as local\_file:

                ftp.retrbinary('RETR ' + ftp\_file, local\_file.write)

            logger.info(f'Successfully downloaded {ftp\_file}')

            return True

        except ftplib.all\_errors as e:

            logger.error(f"Attempt {attempt + 1} failed to download {ftp\_file}: {e}")

            time.sleep(1)  # Wait before retrying

    return False

def lambda\_handler(event, context):

    tmp\_files = os.listdir('/tmp/')

    statuscode = 200

    statusmessage = 'Executed'

    file\_size = 0

    threshold\_file\_size = 52428800  #1024\*1024\*50 (50MB)

    # Get list of files already in the S3 bucket

    s3\_files = list\_s3\_files(s3\_bucket\_name)

    # Connect to FTP server

    ftp = ftplib.FTP(FTP\_HOST)

    try:

        ftp.login(user=FTP\_USER, passwd=FTP\_PSWD)

        ftp.set\_pasv(True)

        ftp.sendcmd('TYPE I')  # Ensure binary mode

    except ftplib.error\_perm as e:

        statuscode = 403

        statusmessage = f"Error {e} in connecting to FTP Server"

        logger.error(f"Error during login or setting binary mode: {e}")

        ftp.quit()

        raise

    # List files in the FTP directory

    try:

        ftp\_files = ftp.nlst()

    except ftplib.all\_errors as e:

        logger.error(f"Failed to list files in FTP directory: {e}")

        ftp.quit()

        raise

    for ftp\_file in ftp\_files:

        # Check if the file already exists in S3

        if ftp\_file in s3\_files:

            logger.info(f'Skipping {ftp\_file} as it already exists in S3')

            continue

        # Download each file with retry mechanism

        local\_filename = f"/tmp/{ftp\_file}"

        if not download\_file(ftp, ftp\_file, local\_filename):

            logger.error(f"Failed to download {ftp\_file} after multiple attempts")

            continue

        # Upload to S3

        try:

            if os.path.exists(local\_filename):

                file\_size = os.path.getsize(local\_filename)

                logger.info(f"Size of the file '{local\_filename}': {file\_size} bytes")

            else:

                logger.error(f"File '{local\_filename}' does not exist")

            if file\_size >= threshold\_file\_size:

                s3\_client.upload\_file(local\_filename, s3\_bucket\_name, ftp\_file)

                logger.info(f'Successfully uploaded {ftp\_file} to S3')

            else:

                logger.error(f"Size of the file '{local\_filename}': {file\_size} bytes is not uploaded to S3 Bucket")

                continue

        except Exception as e:

            logger.error(f"Failed to upload {ftp\_file} to S3: {e}")

            continue

        finally:

            # Remove local file after upload although the tmp files are removed

            if os.path.exists(local\_filename):

                os.remove(local\_filename)

        logger.info("After Removing Files checking in /tmp: %s", tmp\_files)

    # Close FTP connection

    try:

      ftp.quit()

    except Exception as e:

        logger.error(f"Error {e} in Closing the FTP connection")

    return {

        'statusCode': statuscode,

        'body': statusmessage

    }

1. **Adjusting the configuration of Lambda Function**

In the **Configuration** tab, select **General Configuration**. Edit the settings to adjust the memory based on the file size, such as setting it to 250 MB. Also, modify the timeout duration to 10 seconds or more, depending on the processing requirements. Finally, save the configuration.

1. **Creating Environment Variables**

In the **Environment Variables** section, create variables by specifying the key and corresponding value for the FTP Server Host, FTP User, and FTP Password. (The code also illustrates the use of these environment variables.

**Why using Environment Variables is essential?**

Environment variables allow you to keep your function's code separate from its configuration. This means you can change values like FTP Host, FTP User, FTP Password and S3 bucket names (in our code we have not used it as an environment variable but this can also be an example) without modifying the code itself.

By using environment variables, you can avoid hard-coding sensitive information in your code, which can pose security risks if the code is exposed.

It provides **easier testing** by quickly changing environment variables to simulate different scenarios or test with different data without needing to alter the code.

1. **Encrypting the Environment Variables**

AWS provides the option to encrypt environment variables with AWS Key Management Service (KMS), adding an extra layer of security.

Navigate to the **Encryption Configuration** section and check the box to **Enable helpers for encryption in transit**. Next, select **Customer Master Key**.

Open AWS Services and choose **Key Management Services (KMS)**. Create a key by selecting **Customer Managed Key** and retain all the default settings. Provide a name and description for the key. In the **Key Administrator** section, select the role assigned to the Lambda function previously. Finally, click **Create Key**.

Navigate to **Configuration**, then select **Environment Variables**, and proceed to **Encryption in Transit**. Choose the custom key that you created earlier.

Use the code provided in the suggestion next to the values to decrypt in the code. (This information is also included in the code but can be found alongside the key-value pairs as well.)

1. **Testing**

Test the working of Lambda function code, by deploying the changes then hitting Testing.

1. **Creating Event Bridge (Scheduling Triggers)**

After successful deployment of Lambda function we can now create a scheduler for triggering the Lambda function on the incoming of FTP files on daily basis.

Navigate to **Amazon EventBridge** and create a new rule. Specify the schedule expression as cron(0 6 \* \* ? \*). Then, add the scheduler. (You can also review your rule in Amazon EventBridge to see when the trigger will occur, allowing you to make adjustments by editing the event schedule rule as needed.)

With this EventBridge setup, the Lambda function can be tested in the coming days to ensure that the scheduling works correctly.

**MONITORING AND LOGGING**

**CloudWatch Logs** are used to monitor the error handling in lambda function by directly accessing the logs for the Lambda function created above. This provides the debugging of any issue that may occur in Lambda function.

**Cost Monitoring** can be done by analyzing the current month and future prediction of the cost. Tools such as AWS Cost Explorer can be used to monitor and optimize costs over time.

**CHALLENGES**

Several challenges arose while developing this automation:

1. Some FTP files did not conform to the same format as others, which caused issues with the code when accessing files in Snowflake. To mitigate this error, only files larger than 50 MB were uploaded to the S3 bucket.
2. Timeout issues occurred during the execution of the Lambda function, as the timeout settings were not adjusted according to the number of files being retrieved.
3. Errors related to size or memory parameters in the Lambda function were encountered because these settings were not configured based on the memory requirements for the files.
4. To address file size constraints, the Python code needed to be modified, and a try-except block was implemented for better error handling.

**CONCLUSION**

The project successfully created the Lambda Function for automating the transfer of files from FTP to S3 Bucket.

The AWS Lambda function designed to transfer files from an FTP server to an S3 bucket provides an efficient, scalable, and cost-effective solution for automating file transfers in the cloud. By leveraging AWS Lambda's serverless architecture, this approach eliminates the need for managing and provisioning servers, reducing operational overhead and enabling rapid scalability.

Using AWS services such as S3 for storage and IAM for secure access control ensures that data is both reliably stored and protected. The Lambda function’s ability to integrate with other AWS services (such as CloudWatch for monitoring and logging) further enhances its functionality, enabling efficient troubleshooting and management.

Overall, this solution leverages the best practices of AWS cloud computing and can be extended and modified to accommodate additional requirements, such as different file sources, more complex processing, or integration with downstream systems. By implementing this Lambda function, organizations can streamline data workflows, enhance security, and ensure reliable file transfers between their FTP servers and S3 storage.

**APPENDIX**

**Sources**

<https://aws.amazon.com/documentation-overview/lambda/>

<https://aws.amazon.com/de/aws-cost-management/aws-cost-explorer/>

[Course: AWS - Mastering Boto3 & Lambda Functions Using Python | Udemy](https://www.udemy.com/course/mastering-boto3-with-aws-services/learn/lecture/10872496#overview)

<https://pypi.org/project/boto3/>

<https://docs.python.org/3/library/ftplib.html>

<https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html>

<https://aws.amazon.com/de/kms/>

**Glossary**

**AWS (Amazon Web Services)**: A comprehensive cloud computing platform provided by Amazon, offering a wide range of services such as computing power, storage options, and database management.

**FTP (File Transfer Protocol)**: A standard network protocol used for transferring files between a client and a server over a network.

**KMS (Key Management Service)**: A managed service that makes it easy to create and control the encryption keys used to encrypt your data within AWS services.

**IAM (Identity and Access Management)**: A service that enables you to manage user access and permissions to AWS resources securely

**S3 (Simple Storage Service)**: An object storage service offered by AWS that provides scalable, durable, and secure storage for data and files.

**CloudWatch**: A monitoring and observability service in AWS that provides data and insights into the performance and utilization of AWS resources and applications.

**EventBridge**: A serverless event bus service that makes it easy to connect applications using events, enabling you to build event-driven architectures.

**Snowflake**: A cloud-based data warehousing platform that provides capabilities for data storage, processing, and analytics.

**AWS SDK (Software Development Kit)**: A collection of tools and libraries that facilitate the integration of AWS services into applications, allowing developers to interact with AWS resources programmatically.

**Boto3**: The Amazon Web Services (AWS) SDK for Python, which allows developers to write software that makes use of AWS services.

**ftplib**: A Python standard library module that provides tools for implementing the FTP protocol, allowing users to interact with FTP servers for file transfer operations.