Building A Secure File Storage Solution On AWS With Role-Based Access Control

Project Objective

The objective of this project is to design and deploy a secure, scalable, and auditable file storage solution on AWS, fully automated using Terraform Infrastructure as Code (IaC). The system is built to enable different users to upload and download files securely using pre-signed URLs, with access managed through fine-grained IAM roles and temporary credentials issued by AWS Security Token Service (STS).

A key component of the system is support for multi-part upload, allowing users to efficiently upload large files (typically over 20MB) to S3 in smaller chunks. This not only improves performance and reliability but also ensures that interrupted uploads can be resumed.

In addition to secure storage and access, the architecture emphasizes visibility and operational excellence by integrating AWS CloudTrail, Amazon CloudWatch, and Amazon SNS to enable real-time monitoring, logging, and alerting for all S3 operations. Key features such as versioning, encryption (SSE-KMS), and public access blocking are enforced to ensure data integrity and compliance with cloud security best practices.

By using Terraform to provision and manage all AWS resources, the solution ensures repeatability, traceability, and scalability, supporting modern DevOps principles and cloud governance.

Personal Motivation

This project stems from my interest in mastering secure and automated cloud infrastructure provisioning, particularly in real-world scenarios where file storage must be both accessible and strictly governed. I wanted to explore how Terraform can be used to automate not only resource deployment but also enforce security controls and operational monitoring (key pillars in cloud architecture). Through this project, I also deepened my understanding of IAM roles, STS, multi-part uploads, and event-driven monitoring, all of which are crucial skills for any cloud engineer or DevOps professional.

Project Overview

This project implements a secure, role-based file storage system on AWS using IAM, S3, STS, CloudTrail, CloudWatch, and Terraform. Users (Uploader, Viewer, Admin) assume IAM roles via temporary credentials and access files through pre-signed URLs generated by a Python script. S3 securely stores files with encryption, versioning, and blocked public access. All user activity is monitored via CloudTrail and analyzed through CloudWatch Logs and Alarms. Alerts for abnormal

behavior are sent via SNS. The entire infrastructure is provisioned and managed using Terraform, ensuring automation, consistency, and ease of maintenance.

Real-World Motivation

This project was inspired by the increasing need for safe and reliable ways to share important documents (like legal papers, medical records, or financial files) especially in organizations that deal with private information. As someone working toward a career in keeping cloud systems secure and efficient, I wanted to show how it's possible to build a solution that not only protects files, but also keeps track of who accessed them and when. I used trusted tools from Amazon Web Services (AWS) to make sure everything is secure, easy to manage, and ready for real-world use.

Technologies Used & Project Architecture

Technologies Utilized:

- **Infrastructure as code Technology.**
    - Terraform**:** For automating the provisioning and configuration of all AWS resources.
- **Scripting & Automation Technologies**
    - Python: Used to write a script that assumes IAM roles using STS. Generates and uses pre-signed URLs for file uploads (single-part and multi-part) and downloads.
    - Boto3: AWS SDK for Python used in the script to interact with STS and S3.
- **Identity and Access Management Technology**
    - AWS STS (Security Token Service): Used to define roles: Uploader, Viewer, and Admin. Fine-grained permissions control S3 access via policies.
    - IAM Roles (Uploader, Viewer, Admin): Issues temporary credentials for users assuming IAM roles. Used by a script to authenticate and request pre-signed URLs.
- **Storage Technology**
    - S3 Bucket: For storing data.
- **Monitoring, Logging & Alerting**
    - AWS CloudTrail: For Loging API activities (e.g., S3 PUT/GET events) in all AWS services.
    - Amazon CloudWatch: Monitors CloudTrail log files.
    - Amazon SNS Topic: For Sending alert notifications (e.g., via email or SMS) when specific thresholds are met.
- **Security & Encryption Technology**
    - **AWS KMS (Key Management Service)**: For Server-Side Encryption - Key Management Service encryption on the S3 bucket.

Architecture Flow:

Below is the link to the architecture diagram of this project:

https://github.com/philipessel/Secure-File-Storage-In-S3/blob/main/docs/project-architecture-diagram.jpg

Steps To Deploy The Project

The following are the steps to deploy the project

1. Project Planning

- The project was carefully planned around a well-defined objective and scope, focused on building a secure, auditable, and scalable file storage solution on AWS.
- A detailed architecture diagram was developed to visually represent the key AWS services, their interactions, and role-based access controls. This provided a clear roadmap for implementation.
- This planning phase allowed early identification of potential challenges, better resource allocation, and ensured a structured and efficient deployment process using Terraform.

2. Setting Up the Project Environment

To build and run this project successfully, we need to set up our development environment with the right tools and dependencies. Below are the steps followed:

- **Install Terraform**.
  - Terraform is the infrastructure as code tool we will use to automate the creation and management of AWS resources.
  - Download Terraform and follow the installation instructions for your operating system.

- **Install AWS CLI**
  - The AWS Command Line Interface (CLI) will allow us to interact with AWS services directly from your terminal.
  - Download AWS CLI and follow the installation instructions for your operating system.

- **Configure AWS CLI**
  - Once installed, configure it with your credentials by running the command below:

```
aws configure
```

  - You'll be prompted to enter your AWS Access Key, Secret Access Key, Region, and output format.

- **Install Python 3**:
    - Python is the programming language we will use to write and run our automation script. It comes with pip, the package manager for installing additional Python libraries.
    - Download Python 3 and follow the installation instructions for your operating system.

- **Create And Activate A Virtual Environment For The Project**:
    - A virtual environment is like a separate workspace just for this specific project. It lets you install Python packages in that space without changing anything in the main Python setup on your computer. This way, you can use specific versions of packages for one project without messing up others — super helpful when you're working on multiple projects that need different tools or versions.
    - Run the following command to create the virtual environment named `.venv`

```
python -m venv .venv
```

    - Run the following command to activate the virtual environment:

```
.venv\Scripts\activate
```

    - Once activated, all Python packages you install will stay inside our project environment.

- **Install Boto3 Inside The Virtual Environment**:
    - Boto3 is the official Python SDK (Software Development Kit) for AWS. It will let our Python code talk to AWS services like S3, IAM, CloudTrail, etc. To install it, run the command below inside your virtual environment:

```
pip install boto3
```

- **Install Requests Inside The Virtual Environment:**
    - The requests library is a simple, user-friendly, and widely used Python library for making HTTP requests. Specifically in this project, we will use it to perform HTTP PUT requests when uploading each part of our multi-part upload to Amazon S3 via presigned URLs. It therefore important that we install it. To install it, run the following command inside your virtual environment:

```
pip install requests
```

- **Install python-dotenv Inside The Virtual Environment**:
    - The python-dotenv library loads environment variables from a `.env` file into your Python script. Since we will be using `.env` file in this project, we will have to install this package. Using `.env` files will help us to keep sensitive values like access keys and role ARNs separate from our code. To install it, run the following command inside your virtual environment:

```
pip install python-dotenv
```

3. Infrastructure as Code: Defining AWS Resources with Terraform

- The folder structure below represents the `Secure-File-Storage-Solution-project`. Within the `infrastructure/` folder, you'll find all the Terraform configuration files (`.tf files`) that define the AWS resources required for this solution. Each file is modular, serving a specific purpose in organizing and managing the infrastructure efficiently.

```
secure-file-storage-project/
├── docs/
│       ├── project-architecture-diagram.jpg
│       ├── project-overview.md
│       ├── security-policies.md
│       └── setup-guide.md
│
├── .venv/
│
├── infrastructure/
│       ├── provider.tf
│       ├── variables.tf
│       ├── terraform.tfvars
│       ├── admin-role.tf
│       ├── uploder-role.tf
│       ├── viewer-role.tf
│       ├── s3-for-file-storage.tf
│       ├── s3-for-cloud-trail-logs.tf
│       ├── cloud-trail.tf
│       ├── cloud-watch.tf
│       ├── sns.tf.
│       ├── outputs.tf
│       ├── .gitignore
│       ├── .env
│       └── s3-access-script.py
│
└── README.md
```

- Key Terraform Configuration Files (`.tf files`) in the Project Folder:

  The provider.tf File

  - The provider.tf file is a foundational part of this Terraform project. It defines the cloud provider Terraform will use to provision infrastructure. In our case, the project uses Amazon Web Services (AWS) as the cloud platform. You can visit my GitHub Link for the actual code for this configuration file.

  The variables.tf File

  - The variables.tf file is used to declare all the input variables needed by the Terraform configuration. This promotes flexibility, modularity, and ease of reuse across multiple environments or deployments. You can visit my GitHub Link for the actual code for this configuration file.

  The terraform.tfvars File

  - The `terraform.tfvars` file is used to assign values to the input variables defined in the variables.tf file. This file allows you to customize configurations without modifying the core code, making it easy to deploy the same infrastructure across different environments or regions. You can visit my GitHub Link for the actual code for this configuration file.

  The admin-role.tf File

  Below is the list of what the `admin-role.tf` configuration file will do:

  - Creates an IAM Role named using the `admin_role_name` variable.
  - Defines a trust policy that allows a specific IAM user (admin-user) to assume the role using `sts:AssumeRole`.
  - Creates an inline IAM policy that grants the role full access to an S3 bucket, including:

    - Full access to the bucket itself.
    - Full access to all objects within the bucket.

  - Attaches the policy to the IAM role, enabling the trusted user to perform any action (s3:*) on the specified S3 bucket and its contents.
  - You can visit my GitHub Link for the actual code for this configuration file.

  The viewer-role.tf File

  Below is the list of what our viewer-role.tf configuration file will do:

  - Creates an IAM Role named using the viewer_role_name variable.
  - Defines a trust policy that allows a specific IAM user (currently admin-user, but intended to be a viewer user) to assume the role using sts:AssumeRole.
  - Creates an inline IAM policy that grants the role read-only access to objects within a specified S3 bucket:

    - Specifically allows the s3:GetObject action to enable file downloads.

- Attaches the policy to the IAM role, so the trusted user can view (but not modify or delete) the contents of the S3 bucket.
- You can visit my GitHub Link for the actual code for this configuration file.

The uploader-role.tf File

Below is the list of what our uploader-role.tf configuration file will do:

- Creates an IAM Role named using the uploader_role_name variable.
- Defines a trust policy that allows a specific IAM user (currently set to admin-user, but meant for an uploader user) to assume the role using sts:AssumeRole.
- Creates an inline IAM policy that grants the role upload permissions for a specific S3 bucket, including:
    - s3:PutObject – Upload files.
    - s3:InitiateMultipartUpload – Start a multi-part upload.
    - s3:UploadPart – Upload individual parts.
    - s3:CompleteMultipartUpload – Finalize multi-part uploads.
    - s3:AbortMultipartUpload – Cancel multi-part uploads.
- Attaches the policy to the IAM role, enabling the trusted user to upload files (including large files using multi-part upload) to all objects within the specified S3 bucket.
- You can visit my GitHub Link for the actual code for this configuration file.

The s3-for-file-storage.tf File

Below is the list of what our s3.tf configuration file will do:

- Creates the main S3 bucket named using the bucket_name variable.
- Enables versioning on the bucket to maintain multiple versions of objects, allowing recovery from unintended overwrites or deletions.
- Blocks all forms of public access to ensure the bucket and its contents are private and secure.
- Enables server-side encryption using AWS KMS-managed keys (aws:kms) for automatic encryption of data stored in the bucket.
- Defines a CloudTrail-specific bucket policy to:
    - Ensure CloudTrail has permission to get the bucket's ACL for proper access setup.
- Adds bucket policy statements to grant specific roles access:
    - Uploader Role: Granted permission to upload files, including multi-part upload capabilities.
    - Viewer Role: Granted read-only access to download/view objects in the bucket.
- Uses dynamic values (like account ID and role names) via Terraform variables and data sources to make the setup reusable across different environments.
- You can visit my GitHub Link for the actual code of this configuration file.

The s3-for-cloud-trail-logs.tf File

Below is the list of what our s3.tf configuration file will do:

- Creates a dedicated S3 bucket for storing CloudTrail logs named using the variable cloudtrail_logs_bucket_name.
- Enables versioning on the S3 bucket to keep track of changes made to log files.
- Blocks all public access to the S3 bucket to ensure that the logs are not publicly accessible.
- Configures the S3 bucket to use server-side encryption (SSE) with AWS KMS (Key Management Service) to encrypt log files stored in the bucket.
- Create an S3 Bucket Policy that will do the following:

    - Allows the CloudTrail service to write logs to the bucket (s3:PutObject permission).
    - Ensures CloudTrail has permission to get the bucket ACL (s3:GetBucketAcl).
    - Adds a condition that CloudTrail must set the bucket-owner-full-control ACL on the objects it uploads to the bucket.

- You can visit my GitHub Link for the actual code of this configuration file

The cloud-trail.tf File

Below is the list of what our cloud-trail.tf configuration file will do:

- Creates a CloudTrail trail named from var.cloudtrail_name.
- Stores logs in a dedicated S3 bucket (aws_s3_bucket.cloudtrail_logs_bucket).
- Does not include global service events (like IAM, STS).
- Enables multi-region logging across all AWS regions.
- Sends notifications to an SNS topic (aws_sns_topic.security_alerts).
- Sends logs to a CloudWatch log group (aws_cloudwatch_log_group.monitoring_logs).
- Uses an IAM role (aws_iam_role.cloudtrail_to_cloudwatch_role) to allow CloudTrail to publish to CloudWatch Logs.
- Monitors read and write events on the specified S3 bucket (secure_storage).
- Includes management events in the logs.
- You can visit my GitHub Link for the actual code of this configuration file.

The cloud-watch.tf File

Below is the list of what our cloud-watch.tf configuration file will do:

- Creates a CloudWatch Log Group named from the variable var.cloudwatch_name.
- Creates an IAM Role named cloudtrail-to-cloudwatch-role that allows CloudTrail to assume the role.
- Attaches an IAM policy to the role that grants permission to:
    - logs:CreateLogStream
    - logs:PutLogEvents
- Enables CloudTrail to publish logs to CloudWatch Logs using the created role and permissions.
- You can visit my GitHub Link for the actual code of this configuration file.

The sns.tf File

Below is the list of what our sns.tf configuration file will do:

- Creates an SNS topic named using the sns_topic_name variable.
- Sets up an email subscription to the SNS topic. Notifications sent to the topic will be delivered to the specified email address: philipessel2006@gmail.com.
- You can visit my GitHub Link for the actual code of this configuration file.

The outputs.tf File

Below is the list of what our outputs.tf configuration file will do:

- Outputs the S3 bucket name (s3_bucket_name) after deployment. This makes it easy to reference the bucket ID elsewhere or display it for verification.
- Outputs the ARN of the SNS topic (sns_topic_arn). This provides a reference to the SNS topic for alerting or integration purposes.
- Outputs the email address subscribed to the SNS topic (sns_email_subscription). Useful for confirming the notification recipient.
- Outputs the ARN of the Admin IAM Role (admin_role_arn). This allows referencing or verifying the admin role for auditing or integration.
- Outputs the ARN of the Viewer IAM Role (viewer_role_arn). Useful for identifying the role with read-only access to the S3 bucket.
- Outputs the ARN of the Uploader IAM Role (uploader_role_arn). Helps in referencing the role responsible for uploading files to the S3 bucket.
- Outputs the name of the CloudTrail trail (cloudtrail_name). Provides confirmation and reference to the trail capturing AWS API activity.
- Outputs the name of the CloudWatch log group (cloudwatch_log_group_name). Enables visibility into where logs are being collected for monitoring purposes.
- You can visit my GitHub Link for the actual code of this configuration file.

4. Initialize Terraform and Deploying Resources

- After defining our infrastructure using Terraform configuration files, the next step is to initialize our project and deploy the resources. This process involves setting up our working directory, previewing planned changes, and applying those changes to our AWS environment. Follow the steps below to get started:
  - **Initialize Terraform:** This process downloads the necessary provider plugins and sets up the backend for state management. Upon successful initialization, a `.terraform/` directory and a `.terraform.lock.hcl` file will be created automatically in your project folder. The `.terraform/` folder stores downloaded providers and backend setup to enable terraform to talk to AWS.
    The `.terraform.lock.hcl` keeps track of which versions of the providers Terraform downloaded to ensure consistency. Run the command below to initialize the Terraform working directory.

```
terraform init
```

- **Generate Execution Plan:** Run the command below to generate an execution plan. Terraform will automatically create a file named `tfplan`, which contains the planned infrastructure changes. This allows you to review the changes before applying them:

  ```
  terraform plan -out=tfplan
  ```

- **Apply the Execution Plan:** Execute the command below to apply the saved plan and provision the defined resources in your AWS environment:

  ```
  terraform apply "tfplan"
  ```

  Note:

  After running `terraform apply`, `terraform.tfstate` file is automatically created in the project folder. This `terraform.tfstate` file stores the actual state of your infrastructure (what Terraform has deployed in your cloud environment). Every time you apply new changes, this file is updated to reflect the current infrastructure. Another file, `terraform.tfstate.backup`, is created automatically whenever Terraform updates the state file.The `terraform.tfstate.backup` file serves as a backup of the previous state, so if something goes wrong during an update, you can roll back. It should however be noted that if you are using remote state (like in an S3 bucket), these files won't be created in your local project folder. They will be stored remotely instead.

5. Verify Resource Formation in AWS

- After running `terraform apply "tfplan"` to create resources in your AWS cloud, you can follow the steps below to verify if the resources have been created in AWS:

  - Log in to the AWS Console.
  - Navigate to the relevant AWS services (e.g., S3 Buckets, IAM Roles, SNS, CloudTrail).
  - Check if the resources (S3 Buckets, IAM Roles, etc.) are visible and match the configuration you defined in Terraform.

- Alternatively, you can inspect the `terraform.tfstate` file to see the resources that were deployed and their current attributes. To view this in a human readable format, run the command below:

  ```
  terraform show
  ```

6. Create .env File

To keep the project safe and well-organized, sensitive settings like AWS region and user roles are saved in a file called `.env`. This file stores sensitive information, making it useful for separating these sensitive configuration data (like AWS credentials, database URLs, API keys, etc.) from your actual source code.

This helps protect private details and follows best practices for working with cloud projects.

Below is an example of what my .env file will contain in this project:

```
S3_BUCKET_NAME="your-s3-bucket-name"
AWS_REGION="your-aws-region"
UPLOADER_ROLE_ARN="arn:aws:iam::your-account-id:role/your-uploader-role"
VIEWER_ROLE_ARN="arn:aws:iam::your-account-id:role/your-viewer-role"
ADMIN_ROLE_ARN="arn:aws:iam::your-account-id:role/your-admin-role"
```

Note 1: Replace the following with your actual values.

- your-s3-bucket-name
- your-aws-region
- your-account-id
- your-uploader-role
- your-viewer-role
- your-admin-role

7. Create .gitignore File

- At the end of this project, the entire folder will be pushed to GitHub for version control and public sharing. The project directory contains a variety of files and folders, some of which are safe to share publicly, while others are not.

  For example, as stated earlier on, files like `.env` contain sensitive information, such as access keys or environment variables, that should never be exposed.

  Additionally, some files and folders such as `.terraform`, may not contain sensitive data but are too large to upload into GitHub efficiently. Including them in the repository could slow down the project and clutter the version history.

  To avoid accidentally uploading these files, I created a `.gitignore` file in the root of my project. This file lists all the files and folders that should be excluded when pushing the project to GitHub. Below is a snippet of what I included in my `.gitignore` file:

```
# Security
.env

# terraform files
*.tfstate
```

```
*.tfstate.backup
.terraform/

# Python
__pycache__/
*.py[cod]
*.egg-info/
.venv/
env/
venv/
```

This setup helps keep the repository secure, lightweight, and easy to manage, while still sharing all the relevant code and configuration needed to understand and replicate the project.

8. Writing the Python Script

- The core functionality of our Secure File Storage application is implemented in a Python script named s3_access_script.py, which resides within the project folder as shown in the folder structure above. This script handles the logic for securely interacting with Amazon S3 by assuming various IAM roles defined for the application. The s3_access_script.py script implements the following features:
    - Loads environment variables from a .env file for AWS credentials and configuration.
    - Initializes AWS STS and S3 clients using boto3.
    - Assumes 3 different IAM roles (Uploader, Viewer, Admin) using AWS STS.
    - Generates S3 presigned URLs for:
        - Uploading files (single-part).
        - Uploading multi-part file chunks.
        - Downloading files.
    - Supports multi-part file uploads for files ≥ 20MB by:
        - Initiating a multi-part upload.
        - Generating presigned URLs for each part.
        - Uploading parts with `requests.put()`.
        - Completing the multi-part upload after all parts are uploaded.
    - Uploads files directly to S3 using assumed credentials.
    - Downloads files directly from S3 using assumed credentials.

- Prompts user for action input at runtime (upload, download, admin-upload, admin-download).
- Performs the appropriate action based on user input:
  - Upload via presigned URL or direct upload.
  - Download via presigned URL or direct download.
- Uses math to determine chunking logic for multi-part upload.
- Handles errors gracefully using try/except blocks and print helpful error messages.

- You can view the complete implementation of the s3_access_script.py script in my GitHub Repository.

9. Testing the Application (Running Python Script)

- Our python script contains the logic that allows Security Token Services (STS) to assume these three main roles;
  - Uploader-Role: This role allows a user to upload files into S3 bucket through pre-signed url. Single-part upload will apply when files are less than 20MB but multi-part upload will apply when files are above 20MB.
  - Viewer-Role: This role allows a user to download files from S3 bucket through pre-signed url.
  - Admin-Role: This role gives a user full access and allows direct upload of files into and direct download of files from S3 bucket without using pre-signed url.
- So, we would basically test whether these functionalities are working.
- In VS Code, open your Powershell terminal.
- Navigate to the directory where you saved the `s3-access-script.py` file.

1. **Test Upload via Presigned URL (Uploader-Role) – Single-part upload:**

   Note: Use files that are less than 20MB for single-part upload. Also note that the uploader can only upload into the S3 bucket through presigned url.
   - Run the script: `python s3-access-script.py`
   - A list of actions will be displayed (upload/download/admin-upload/admin-download).
   - Input `upload` and press enter.
   - When prompted to enter filename in S3: Input the filename (e.g., `test_upload.txt`) you want to upload into S3 and press enter.
   - When prompted to enter the path to the local file to upload: Input the path to the file location (e.g., `C:\Users\user\Desktop\confusion.docx`) and press enter.

- The script will output a presigned URL. Copy this URL.
- Open a new terminal window and use `curl` to upload your local file as shown below:

```
curl.exe -X PUT -T \path\to\your\local\file "PRESIGNED_URL"
```

Note:

1. Replace `\path\to\your\local\file` with the actual path of your file
2. Replace `PRESIGNED_URL` with the actual presigned URL generated by the script.

Example is as shown below (the link is the presigned url generated):

curl.exe -X PUT -T C:\Users\user\Desktop\tex_upload.txt "https://secure-file-storage-bucket-philip.s3.amazonaws.com/confusion.docx?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=ASIATLBWZHK34ZKMIFZU%2F20250526%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20250526T073401Z&X-Amz-Expires=3600&X-Amz-SignedHeaders=host&X-Amz-Security-Token=FwoGZXIvYXdzEIn%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaDMP%2FBgOAN93thd7yNSK4AXaOKr4aombSzHC9wcqkpUGbHhYvQ5mXjrmYZfo0CKq1cAyWMPNuvlCH8fsFY%2FXKVAIXkWutu0OMrnmoEpV6iDpwj86Q4xOFtY4yi2mJ%2FLWU8YEaW%2BcHi4blYIzAbXBdspVzMkX2ZBc78lEKEgJObiEsJuYhevUT5obH52Dn9tGcaNceGYtGJSImwsPt6ceDi8bVlgvTsMu2rJSUx4x8AInil175PMqGUD%2BmaTE2S9j2qb2ePBS5I74otLLQwQYyLeUYic3DcRB99uZ%2FvG4XJBMEz9%2FVQ1%2FrMMIP3nl9EznHyZHW6YKpzIBXgK8b2g%3D%3D&X-Amz-Signature=fc645859b7ca30dbca57ddc4f5931f20fbce3b3c42e4cd1961880b622d5ffc55"

- Go to your S3 bucket in the AWS Management Console and verify that the file has been uploaded.

2. **Test Upload via Presigned URL (Uploader-Role) – multi-part upload:**

Note: Use files that are more than 20 MB for multi-part upload.

- Run the script: `python s3-access-script.py`
- A list of actions will be displayed (upload/download/admin-upload/admin-download).
- Input `upload` and press enter.
- When prompted to enter filename in S3: Input the filename (e.g., `video.mp4`) you want to upload into S3 and press enter.
- When prompted to enter the path to the local file you want to upload: Input the path to the file location (e.g., `C:\Users\user\Desktop\video.mp4`) and press enter.

- Multi-part upload will begin and when completed the ETag of the uploaded file will be generated and outputted.
- Go to your S3 bucket in the AWS Management Console and verify that the file has been uploaded.

3. **Test Download via Presigned URL (Viewer-Role):**

   Note: Viewer can only download from the S3 bucket through presigned url.

   - Run the script: `python s3-access-script.py`
   - A list of actions will be displayed (upload/download/admin-upload/admin-download).
   - Input `download` and press enter.
   - When prompted to enter filename in S3: Input the filename (e.g., `test_upload.txt`) you want to download from S3 and press enter.
   - The script will output a presigned URL. Copy this URL.
   - Open a new terminal window and use `curl` to download file from S3 as shown below:

     ```
     Curl.exe "PRESIGNED_URL" -o downloaded_file
     ```

     Note:
     1. Replace `PRESIGNED_URL` with the actual presigned URL generated by the script.

     Example is show is as shown below (the link is the presigned url generated):

     curl.exe https://secure-file-storage-bucket-philip.s3.amazonaws.com/test_upload.txt?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=ASIATLBWZHK3U22HSNIY%2F20250526%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20250526T112500Z&X-Amz-Expires=3600&X-Amz-SignedHeaders=host&X-Amz-Security-Token=FwoGZXIvYXdzEI3%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaDAGKsukjb-VIA9D3eWyK4AUw3BtXhsECyir%2Fc5hGr5EIAGJ1ruhzC0aY1fMmcPtfUvfPd1s2iC6Gy40PvAnjl%2F9z09w2YefHr3hGXoI6RWVccCdBnQCz6jhD5vPfkXP%2FwcynWhyVUGbDRcLlh%2BUulHj7bFJp2WTY8KhRK-KIiYWZoOfqIPa9vijrhTa2HVTkaGY8riXYvoW9UK8UB-fhqyUyLf%2FQfAnrW8EOn9W1Qgsh5xtU56Ig24dyc2OpVHqK-SKzqf0eucq%2BWyIoiZ%2FRwQYyLbjsUX2fi2zP3iltrUJBfEObujVZnJt8Ei-eHKNKUAUB3Lfn%2FH7lyr%2FfG%2FMRbiw%3D%3D&X-Amz-Signature=a16c41170499b192f5e3ebc87e0e75a9a968e40c1e9002c3fdbcb5240a053a98 -o downloaded_file

   - The downloaded file will be stored in the same directory where the python script (s3-access-script.py) is located. To verify that the file has been downloaded, open the project folder and navigate into the directory that contains the python script. Within

this directory, you will find the downloaded file (downloaded_file). Double-click it to open.

4. **Test Direct Upload (Admin-Role):**

   Note: Because Admin has full access to the S3 bucket, upload was direct and there is no need for presigned url. Also, note that the admin-upload function in my python script does not explicitly managing multipart upload steps but rather leverages the boto3 library's upload_file method. This method automatically handles multipart uploads internally for files exceeding a certain size threshold (typically 5GB by default, though configurable). Therefore, even without explicit code for multipart logic, boto3 will implicitly perform a multipart upload when the file size warrants it.

   - Run the script: `python s3-access-script.py`
   - You will be prompted for a list of actions (upload/download/admin-upload/admin-download).
   - Input `upload` and press enter.
   - When prompted to enter filename in S3: Input the filename (e.g., `test_upload.txt`) you want to upload into S3 and press enter.
   - File will be uploaded directly into S3.
   - To verify that the file has been uploaded, go to your S3 bucket in the AWS Management Console and verify that the file has been uploaded.

     .

5. **Test Direct Download (Admin-Role):**

   Note: Because Admin-Role has full access to the S3 bucket, download was direct and there is no need for presigned url.

   - Run the script: `python s3-access-script.py`
   - You will be prompted for a list of actions (upload/download/admin-upload/admin-download).
   - Input `download` and press enter.
   - When prompted to enter filename in S3: Input the filename (e.g., `test_upload.txt`) you want to download from S3 and press enter.
   - File will be downloaded directly and stored in the same directory where the python script (s3-access-script.py) is located. To verify that the file has been downloaded, open the project folder and navigate into the directory that contains the python script. Within this directory, you will find the downloaded file (downloaded_file). Double-click it to open.

Challenges Faced and Lessons Learned

Although the project was an exciting and insightful experience, it came with its own set of challenges. Each challenge provided valuable learning opportunities that contributed to a deeper understanding of AWS services and best practices.

1. **Multi-Part Upload Failed Due to Invalid Chunk Size**

   Challenge: My multipart upload failed or behaved strangely because I set the chunk size to 2MB, which is below the AWS minimum size for S3 multipart uploads.

   What I did to fix it: I updated the script to use a chunk size of at least 5MB, which is the minimum allowed by AWS for multipart uploads.
   .
   Lesson Learned: AWS services have specific limits (like chunk size), and it's important to follow them. Always check the documentation when working with such features.

2. **Getting Alerts to Work**

   Challenge: set up CloudWatch alarms to monitor activity in my S3 bucket, but the alarms were not sending any notifications, even when I expected them to.

   What I did to fix it: I checked the CloudWatch metric filters and noticed they weren't matching the actual log events. I fixed this by updating the filter patterns to match the real error messages that appear when something unusual happens (like a failed upload). Then, I connected the alarms to an SNS topic so they could send email alerts.
   To test if the setup was working, I purposely tried to upload a file using a wrong IAM role (one without permission). This created an error in the logs. CloudWatch picked up the error, and I received an alert email confirming it worked.

   Lesson Learned: For CloudWatch alarms to work, your filter patterns must match real log events exactly. And to be sure alerts are working, it's helpful to simulate common errors so you can see how the system responds.1

3. **Upload to S3 Failed Due to Encryption Settings**

   Challenge: When I tried uploading a file to an S3 bucket using a pre-signed URL, it failed because the bucket was encrypted with AWS KMS, and the upload didn't meet the required Signature Version 4.

   What I did to fix it: I updated the script that generated the pre-signed URL to use Signature Version 4, which is required when uploading to a KMS-encrypted bucket.

   Lesson Learned: AWS encryption settings affect how uploads work. When using KMS, always make sure your requests are signed using Signature Version 4.

4. **Error Assuming IAM Role Due to Missing Trust Policy and Permissions**

Challenge: My Python script couldn't assume an IAM role because the trust policy was missing, and the IAM user didn't have permission to call sts:AssumeRole.

What I did to fix it: I added a trust policy to the IAM role to allow assumption by my user, and also gave my IAM user permission to assume the required roles.

Lesson Learned: For IAM role assumption to work, both the trust relationship and user permissions must be correctly set. Missing either will cause access issues.

5. **Pylance Import Errors in VS Code Due to Missing Libraries**

Challenge: VS Code was showing errors like "Import could not be resolved" for libraries like boto3, requests, and dotenv.

What I did to fix it: I activated my virtual environment and installed the missing libraries using pip. Then I restarted VS Code to refresh the Pylance analysis.

Lesson Learned: Always install required libraries inside your virtual environment, and restart your editor so tools like Pylance can detect them.

6. **Python-dotenv Was Not Working for Environment Variables**

Challenge: My .env file wasn't working in the Python script, and the environment variables weren't being picked up, even though there were no errors.

What I did to fix it: I added the missing lines in my script: from dotenv import load_dotenv and load_dotenv(). This allowed Python to load the variables from the .env file properly.

Lesson Learned: Installing a library is not enough—you must also initialize it correctly in your code. Small missing details like this can cause silent failures.

7. **Missing IAM Role ARNs in Python Script for Role Assumption**

Challenge: The script quietly failed to assume IAM roles because I didn't define the actual ARNs of the Uploader, Viewer, and Admin roles.

What I did to fix it: I added the ARNs directly into the script using environment variables and referenced them in the assume_role() function.

Lesson Learned: Clearly defining and using the right ARNs is essential when working with IAM and STS. Missing variables can break the whole workflow.

Conclusion

This project successfully built a secure and monitored file storage system using AWS. I learned how to keep files safe with S3, control who can access them with IAM roles, track all activity with CloudTrail, and get alerts with SNS. Using Terraform helped me set up everything consistently.

Dealing with various challenges, like getting permissions just right and making sure CloudTrail logged everything, taught me a lot. This project significantly improved my skills in AWS automation and building secure cloud solutions. It's a strong example of my ability to create real-world applications in the cloud.