Term Project Report

Project Option: Development of a Secure and Privacy-Preserving System for AWS or other cloud platform

Team Members:

- Kaden Wingert
- Casper Run
- Alex Elsner
- Ethan Comiskey

Topic: Cloud Storage System Using AWS KMS and S3

Overview:

We developed a secure and privacy-preserving cloud storage system for our term project utilizing AWS services, including S3, Lambda, API Gateway, and AWS KMS. Our solution enables users to upload and retrieve images through a simple web interface while ensuring data confidentiality through encryption and secure access patterns.

Problem Definition:

As cloud computing and storage continues to grow, protecting user data and information has become increasingly important and more complex. Although most cloud providers offer encryption options for storage, users tend to struggle with the technical complexities of such processes. This gap in knowledge and experience leads to security risks, which is incredibly damaging for sensitive personal data such as images, which are often stored without adequate protection.

This project creates a user-friendly and secure cloud storage solution that automatically encrypts/decrypts the user's images. The user does not need to have any knowledge of cryptography to use this solution. Other modern solutions are either too cryptographically complex for users to understand or do not ensure proper encryption practices. The system model being implemented is a user interacting with a cloud-based image storage system via web interface without needing direct access to keys or knowledge of AWS services. AWS Key Management Service (KMS) and S3 are utilized in this process for key management and storage.

The first goal of our implementation phase is to automatically encrypt user images before upload and decrypt when the user downloads from the storage system. Next is to use the KMS to generate and manage the user's keys. Another goal is to create a user-friendly UI that is easy

for any basic user to upload and download without having to interact with the KMS or S3. Through these implementation goals, we aim to develop an easy to use secure cloud image storage system that ensures end-to-end security.

Conceptual Design:

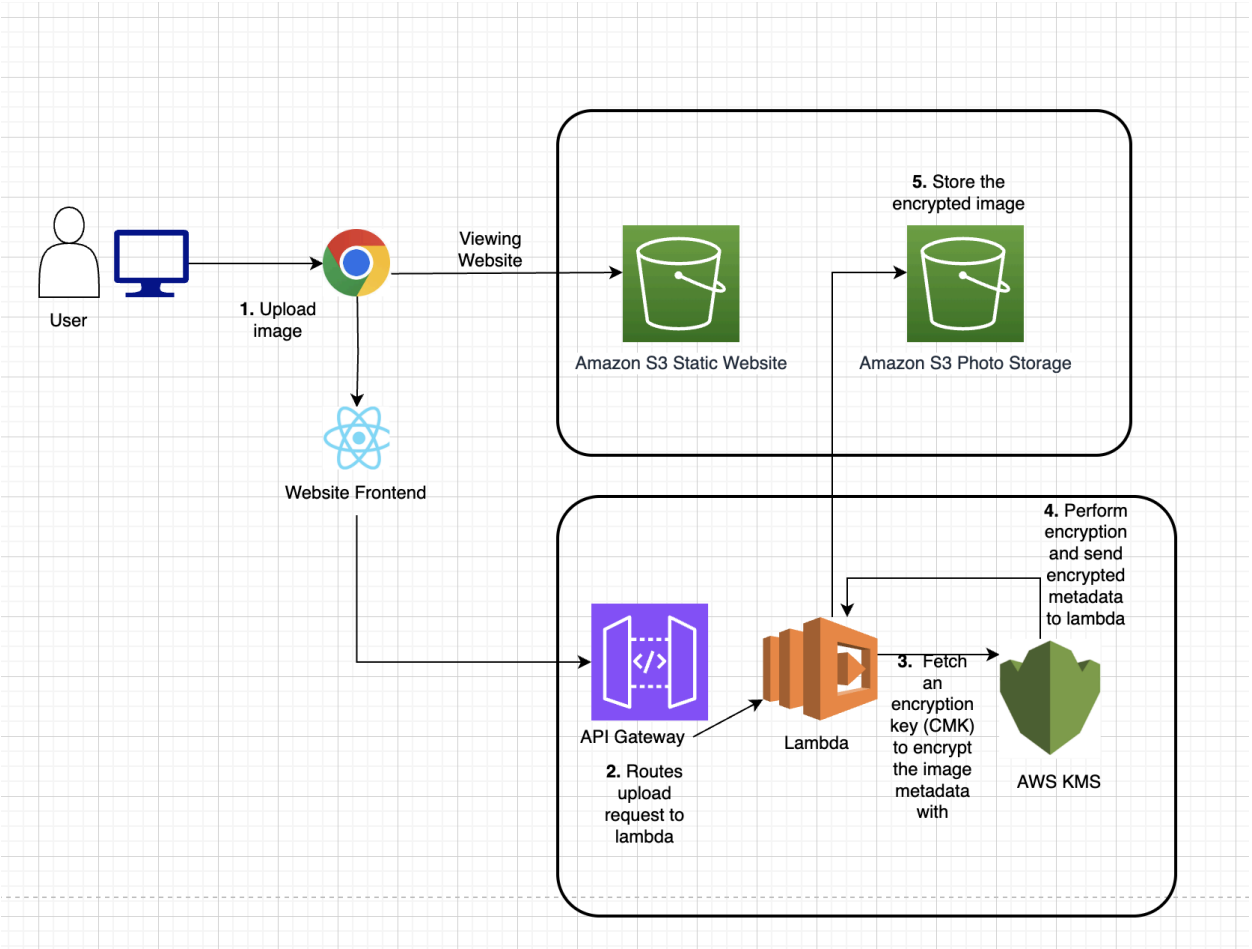

Figure 1: AWS Architectural diagram for our secure image storage system

System Overview

The system is composed of three main components:

Frontend (Client Interface)

- A user-facing web application for uploading and viewing image files.
- The user uploads image files to the backend which are displayed with associated metadata and timestamps.

Backend (Business Logic Layer)

- Includes a serverless lambda function that handles image processing, metadata encryption, and storage/retrieval.
- Provides RESTful API endpoints for frontend interactions.
- Generates pre-signed URLs for secure image access.
- Storage and Security Layer (Data Layer)
- Amazon S3 for storing image files and encrypted metadata.
- AWS KMS for encryption/decryption of sensitive data. KMS uses the AES-256-GCM encryption algorithm. Information about KMS was obtained from Amazon's documentation page titled, *AWS Key Management Service (KMS)*.
- IAM for defining fine-grained access controls across services.

Component Interfaces and Interactions

- User uploads an image via the frontend.
- The image and its metadata are sent to the backend via an HTTP POST request.
- The backend encrypts the metadata with AWS KMS.
- The image and metadata are stored in Amazon S3.
- For retrieval, the backend generates a pre-signed URL and returns it to the frontend for display.
- IAM policies ensure each component only has the permissions it needs.

Design Principles

- Security by design: All sensitive data is encrypted at rest using KMS; access to data is tightly controlled.
- Serverless architecture: Reduces cost and increases scalability.
- Least privilege access control: IAM policies enforce scoped permissions.
- Efficient image access: Pre-signed URLs enable temporary, secure access without exposing the S3 bucket directly. To understand how these urls function, we referenced the following article: *Sharing objects using presigned URLs.*

Implementation description

This section describes the tools, libraries, data structures, and key methods used to actually build the system.

Infrastructure Provisioning with AWS CDK:

The entire backend infrastructure — including Lambda functions, API Gateway routes, S3 buckets, and IAM roles — is provisioned using the AWS Cloud Development Kit (CDK), enabling us to manage our cloud resources through code. This ensures repeatable

deployments, version control, and easier maintenance across development stages. The *AWS Cloud Development Kit (CDK) v2 API Reference* documentation was referenced when constructing the cdk

Frontend:

The frontend is a React-based single-page application hosted on an Amazon S3 bucket configured as a static website. To develop the frontend, the react documentation page, *React – A JavaScript library for building user interfaces,* was referenced. It provides a clean UI for:

- Uploading image files
- Displaying stored images
- Viewing timestamps and image metadata
- It communicates with a backend API for all operations involving S3.

Backend:

The backend is implemented with the following technologies:

- **AWS Lambda**: Acts as a serverless compute layer for handling CRUD operations on images. This allows us to not have to pay for a constantly-running ec2 instance and instead only pay for the compute time the lambda function actually uses.
- **API Gateway**: Provides a RESTful API interface for the frontend to interact with Lambda functions.
- **Amazon S3**: Two S3 buckets are used — one for storing encrypted image files and another for hosting the frontend static website.
- **AWS KMS**: Manages encryption and decryption keys used for securing image metadata.

Security and Privacy Mechanisms:

Encryption via AWS KMS:

- Before storing image files, encrypted metadata is generated using AWS Key Management Service (KMS)
- The metadata includes the filename and other details, which are serialized and encrypted using a KMS Customer Master Key (CMK).
- This encrypted metadata is stored alongside the image in S3, ensuring sensitive information is protected at rest.

```python
def create_kms_key_policy(self):
        return iam.PolicyDocument(
            statements=[
                iam.PolicyStatement(
                    sid="AllowKMSAdministration",
                    effect=iam.Effect.ALLOW,
                    principals=[iam.AccountRootPrincipal()],
                    actions=["kms:*"],
                    resources=["*"]
                ),
                iam.PolicyStatement(
                    sid="AllowLambdaToUseKey",
                    effect=iam.Effect.ALLOW,
                    principals=[iam.ServicePrincipal("lambda.amazonaws.com")],
                    actions=[
                        "kms:GenerateDataKey",
                        "kms:Encrypt",
                        "kms:Decrypt",
                        "kms:DescribeKey"
                    ],
                    resources=["*"]
                )
            ]
        )
```

Figure 2: CDK code for creating the policy for KMS. This gives the permissions to encrypt and decrypt the key

```python
def retrieve_images():
    try:
        response = s3_client.list_objects_v2(Bucket=bucket_name)

        if "Contents" not in response:
            return {
                "statusCode": 200,
                "headers": cors_headers,
                "body": json.dumps({"images": []})
            }

        images = []
        for obj in response["Contents"]:
            file_key = obj["Key"]
            presigned_url = s3_client.generate_presigned_url(
                "get_object",
                Params={"Bucket": bucket_name, "Key": file_key},
                ExpiresIn=3600  # URL valid for 1 hour
            )

            images.append({
                "key": file_key,
                "imageUrl": presigned_url,
                "lastModified": obj["LastModified"].isoformat()
            })

        return {
            "statusCode": 200,
            "headers": cors_headers,
            "body": json.dumps({"images": images})
        }

    except ClientError as e:
        logger.error(f"Failed to list images: {e}")
        return {
            "statusCode": 500,
            "headers": cors_headers,
            "body": json.dumps({"error": "Failed to retrieve images"})
        }
```

Figure 3: Lambda code for retrieving images

Figure 4: Lambda code for uploading images

```python
def get_signed_url(file_key):
    try:
        signed_url = s3_client.generate_presigned_url(
            "get_object",
            Params={"Bucket": bucket_name, "Key": file_key},
            ExpiresIn=3600  # URL expires in 1 hour
        )
        logger.info(f"GENERATED SIGNED URL: {signed_url}")


        return {
            "statusCode": 200,
            "headers": cors_headers,
            "body": json.dumps({"signed_url": signed_url})
        }

    except Exception as e:
        logger.error(f"Failed to generate signed URL: {e}")
        return {
            "statusCode": 500,
            "headers": cors_headers,
            "body": json.dumps({"error": str(e)})
        }
```

Figure 5: Lambda code for getting the signed url for an image

Fine-Grained Access Control with IAM:

- IAM policies were used to enforce the principle of least privilege:
    - The Lambda function was granted only the permissions necessary to `put`, `get`, and `delete` objects in the S3 bucket.
    - It also has scoped permissions for `Encrypt`, `Decrypt`, and `GenerateDataKey` operations using the CMK.
    - Access to invoke the Lambda function from API Gateway is restricted to the `apigateway.amazonaws.com` service principal.

- These permissions were defined and provisioned using AWS CDK, ensuring a clear and auditable infrastructure-as-code setup.

Image Storage:

- Image files are base64-encoded and uploaded via a POST request to our Lambda function.
- Files are stored securely in the designated S3 bucket with attached encrypted metadata.

Image Retrieval with Pre-Signed URLs:

- To view images securely, we use pre-signed URLs that are generated on-demand via the Lambda function.
- These URLs grant temporary access (1-hour expiration) to the S3 object, ensuring that direct access to the bucket is not possible.
- The frontend fetches these signed URLs for each image and displays them accordingly.
- Without pre-signed URLs, Lambda would need to download from S3 and then return the image data, which is bad for performance and cost).
- Since this project is more of a proof of concept, anyone can view the images. In a real environment, the pre-signed urls would be used to only show certain images to authorized users

Lambda Logic (Backend Highlights):

The Lambda function routes requests based on the HTTP method:

- **POST `/images`**:
  - Decodes and stores base64 image data.
  - Encrypts metadata with AWS KMS.
  - Uploads the image to S3 along with encrypted metadata.

- **GET `/images`**:
  - Lists all images stored in the bucket.
  - Returns pre-signed URLs for secure viewing.

This design ensures minimal attack surface and fine-grained access control using AWS IAM roles and temporary URL generation.

Frontend Features:

The React application handles:

- **File selection and upload** using `FileReader` to convert the file into base64 format.

- **Display of images** by consuming the `/images` endpoint and rendering pre-signed URLs.

Self-evaluation

- Automatic Encryption/Decryption: The system automatically encrypts metadata (filenames, etc.) using AWS KMS (AES-256-GCM) before storing images in S3. This means that the users never handle keys directly, ensuring end-to-end encryption without requiring cryptographic knowledge.
- Secure Key Management with KMS: AWS KMS manages the symmetric CMK, enforcing automatic key rotation and access control via IAM. Keys are never exposed to users or stored insecurely, mitigating risks like key leakage or misuse.
- User-Friendly Interface: The React frontend hides all cryptographic and cloud complexities behind a simple UI for upload/download.
- Performance and Cost Efficiency:Pre-signed URLs avoid unnecessary Lambda compute usage for image retrieval, reducing costs. The CDK also allowed us to only have the infrastructure provisioned when we needed it, preventing any unnecessary costs
- Least-Privilege Access Control IAM policies restrict Lambda to only essential S3/KMS operations, and API Gateway locks down Lambda invocations.

Areas for Improvement
Fine-Grained Authorization: The current design allows any user to view all images. A production system would need user authentication (e.g., Cognito) to scope pre-signed URLs to individual users.

User Guide

If a developer were to set up their local environment from scratch, the following are the necessary steps which can also be viewed in the README.md of the repository:

Prerequisites

1. You must have the aws cli installed. For instructions on how to do so, visit this link: https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html

2. Create a secret access key for your user in the aws console and save both the secret key and the secret access key for later. This allows for authenticating AWS API requests

3. Install the aws cli plugin in vscode, intellij, or whatever IDE you are using

4. Go to the plugin you just installed and click "connect to AWS" and click "add new connection". Then click "IAM credentials" and enter in your user name, access key, and secret access key

5. Now you should be able to run aws commands. If you made this account the default profile, you can run the commands normally. If you named your profile something else, ex: johnDoe, then instead of running a command that starts with 'cdk' such as cdk deploy, you would run cdk deploy --profile johnDoe

Set up the environment

```
python3 -m venv .venv   # Create virtual environment

source .venv/bin/activate   # Activate it (Linux/Mac)

 or .venv\Scripts\activate.bat (Windows)

 pip install -r requirements.txt
```

Bootstrap your AWS account (first time only):

```
 cdk bootstrap aws://945839052165/us-east-1
```

Deploy the stack

```
cdk deploy --all --require-approval never
```

Upload the Correct Url for API Gateway

AFTER the stack finishes deploying, run this script (which is located in the home directory of this project) to upload the API Gateway URL to the config.json file: `write_config.py`

Destroy the stack

To save costs, destroy the stack after you are done working on it by running `cdk destroy --all  --force`

Frontend Development

Note that if you are ONLY doing changes on the frontend ui, you can cd to the website_assets directory and run `npm start` because the frontend is just a basic react app

Other

Since this code uses AWS CDK, do NOT create/delete/modify resources from the console. Doing so will create what is called 'stack drift' and will mess things up, and can cause some headaches when trying to recreate/deploy the infrastructure.

Conclusion:

Our project successfully demonstrates how cloud-native services like AWS Lambda, S3, KMS, and API Gateway can be combined to create a secure and privacy-respecting image storage and access system. We ensured user data was protected in transit and at rest by implementing encryption, secure metadata handling, and pre-signed URLs. Additionally, we used AWS Cloud Development Kit (CDK) to provision and manage the infrastructure as code, allowing us to deploy and update our resources in a repeatable, maintainable, and automated manner.

We built a scalable foundation that can easily be extended to support user authentication, tagging, advanced metadata search, and image transformations while maintaining a strong security posture.

References

Amazon Web Services, Inc. (n.d.). *AWS Cloud Development Kit (CDK) v2 API Reference*. Retrieved April 22, 2025, from https://docs.aws.amazon.com/cdk/api/v2/

Amazon Web Services, Inc. (n.d.). *Sharing objects using presigned URLs*. Amazon S3 User Guide. Retrieved April 22, 2025, from https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html

Amazon Web Services, Inc. (n.d.). *AWS Key Management Service (KMS)*. Retrieved April 22, 2025, from https://aws.amazon.com/kms/

Meta Platforms, Inc. (n.d.). React – A JavaScript library for building user interfaces. Retrieved April 22, 2025, from https://react.dev/

Contributions

Kaden - I designed the system's architecture and created the architectural diagram on draw.io. I also developed the code for the CDK for the system's infrastructure so we would be able to set up and tear down the project without incurring constant costs when it is not being used. Lastly, I created the frontend website UI where the user can upload an image and the cards for displaying the images.

Casper -

Ethan -

Alex -