CLOUD COMPUTING

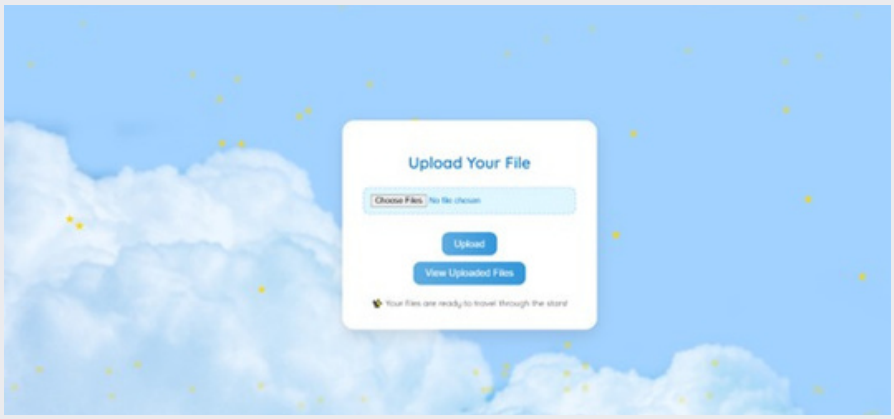
REPORT

Loubid Emad Mohamed 23012047
Marwan Mohamed Zein 23011523
Huda Ali Foad 23011605
Habiba Hamdy Ali 23011064
Mariam Khaled Ahmed 23011528
Karim Mahmoud Elsayed 23011126

Introduction

The goal of this project is to build a simple web application that enables users to upload and download files through a web browser. The core idea is to create a cloud-based system using AWS services such as EC2, S3, IAM, and VPC to host the application, store uploaded files securely, and control access permissions. This project helps students understand the basics of cloud computing and gain practical experience in deploying web applications in a secure and scalable way. The application allows users to upload multiple files, stores them safely in an Amazon S3 bucket, and provides a direct download link for each uploaded file.

This report presents all the steps our team followed to implement this solution successfully. Each section explains a specific part of the process, from creating the AWS account to writing the backend and frontend code, setting up networking, and final testing.



Step 1: Creating an AWS Account

The first step was to set up an AWS account. One team member was responsible for this task. The process involved the following steps:

1. Navigating to the AWS Free Tier page and signing up.
2. Entering personal and payment information (AWS requires a valid credit/debit card).
3. Verifying the identity through a phone number and receiving an OTP.
4. Selecting the "Free Tier" plan to ensure we stay within usage limits to avoid unexpected charges.
5. Completing the signup process and gaining access to the AWS Management Console.

Once the account was ready, the team was able to access all the AWS services needed for the project.

```
import path from 'path';
import { fileURLToPath } from 'url';

const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

export const NotFound = (req, res, next) => {
    res.status(404).sendFile(path.join(__dirname, '../public', 'error.html'));
};

export const Invalid = (err, req, res, next) => {
  if (err.status === 400) {
    return res.status(400).json({ error: 'Invalid request' }); // send JSON or plain text
  }
  next(err);
};
```

Step 2: Launching EC2 Instance and Configuring IAM

To host our web application, we used an Amazon EC2 instance. These are the steps we followed:

1. EC2 Launch:
   - We launched an EC2 instance using Amazon Linux 2 (free tier eligible).
   - Selected t2.micro instance type.
   - Configured instance details and added a key pair for SSH access.
2. Security Groups:
   - Created a new security group to allow HTTP (port 80) and SSH (port 22).
   - Ensured access was limited to known IPs for security.

3. IAM Role Creation:
   - Created a new IAM role named EC2S3AccessRole.
   - Attached the AmazonS3FullAccess policy to this role to allow the EC2 instance to upload files to S3.
   - Linked this IAM role to the EC2 instance during creation.
4. SSH Access and Configuration:
   - Connected to the EC2 instance via SSH.
   - Installed necessary packages like Git and Node.js (detailed in later sections).

This configuration allowed us to securely interact with Amazon S3 from our EC2 instance.

```
import express from 'express'
import path from 'path'
import { fileURLToPath } from 'url'

const router = express.Router();
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

router.get('/',(req,res)=>{
    return res.sendFile(path.join(__dirname,'../public','HomePage.html'))
});

export default router;
```

Step 3: Creating and Configuring Amazon S3 Bucket

We created a new S3 bucket to store uploaded files. The following configurations were applied:

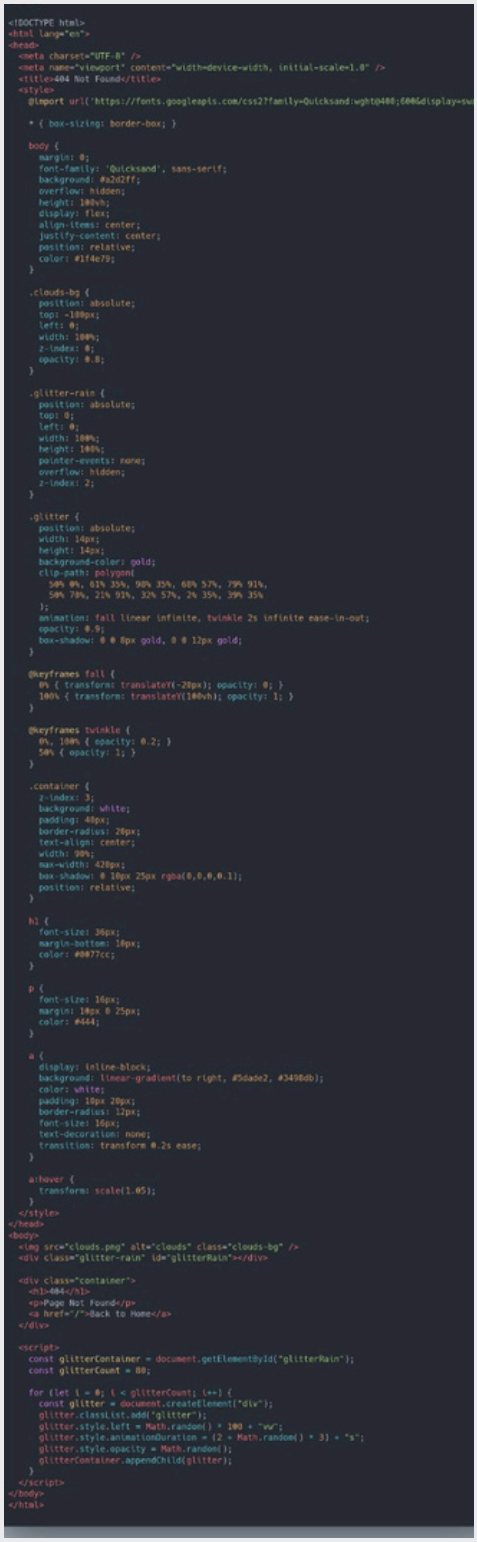
1. Bucket Creation:
   - Named the bucket file-sharing-bucket (globally unique).
   - Chose the same AWS region as the EC2 instance for better performance.
2. Permissions and Access Settings:
   - Enabled block public access (recommended for security).
   - Created a bucket policy to allow access only from the EC2 instance with the attached IAM role.
   - Configured CORS settings to allow cross-origin requests from our frontend.
3. Storage and Folder Structure:
   - We did not create separate folders per user to avoid complexity, since that would require a database and user authentication system.

Now, the bucket was ready to receive and serve files uploaded via the web application

```
import { S3Client, PutObjectCommand, ListObjectsV2Command } from '@aws-sdk/client-s3';
import dotenv from 'dotenv';
import fs from 'fs';
import path from 'path';

dotenv.config();

const s3 = new S3Client({
  region: process.env.AWS_REGION,
  credentials: {
    accessKeyId: process.env.AWS_ACCESS_KEY,
    secretAccessKey: process.env.AWS_SECRET_KEY
  }
});

export const uploadToS3 = async (file) => {
  const fileStream = fs.createReadStream(file.path);
  const uploadParams = {
    Bucket: process.env.AWS_BUCKET_NAME,
    Key: file.filename,
    Body: fileStream,
    ContentType: file.mimetype
  };
  await s3.send(new PutObjectCommand(uploadParams));
};

export const listS3Files = async () => {
  const listParams = {
    Bucket: process.env.AWS_BUCKET_NAME,
  };
  const data = await s3.send(new ListObjectsV2Command(listParams));
  return data.Contents || [];
};
```

Step 4: Web Application Development and S3 Integration

S3 Integration
Backend Implementation:
We used Node.js to create a simple file-sharing server. Here are the key features and steps:
1. File Upload Handling:
o The backend accepts up to 10 files per upload request.
o Before uploading to S3, the server checks each file's extension to ensure it is allowed (e.g., .txt, .pdf, .png).
o Filenames are randomized to prevent name conflicts. Each file name is a 32-character alphanumeric string.
2. Security Measures:
o Rejected potentially dangerous files like .js, .exe, and other executable code.
o Limited accepted MIME types.
3. Logger Middleware:
o Implemented middleware to log upload attempts and status. This helps in debugging and monitoring.
4. Deployment on EC2:
o Installed Node.js using NVM.
o Cloned the GitHub repository with our app code.
o Configured a systemd service to run the app on startup.
Frontend Implementation:
The frontend was built using HTML, CSS, and JavaScript:
• A styled form allows users to select up to 10 files.
• Files are validated on the client side before upload.
• Upload progress is displayed using JavaScript.
• After upload, download links are displayed for each file.
This combination allowed users to interact with the app in a smooth and user-friendly way.



Step 5: Setting Up Networking (Amazon VPC)

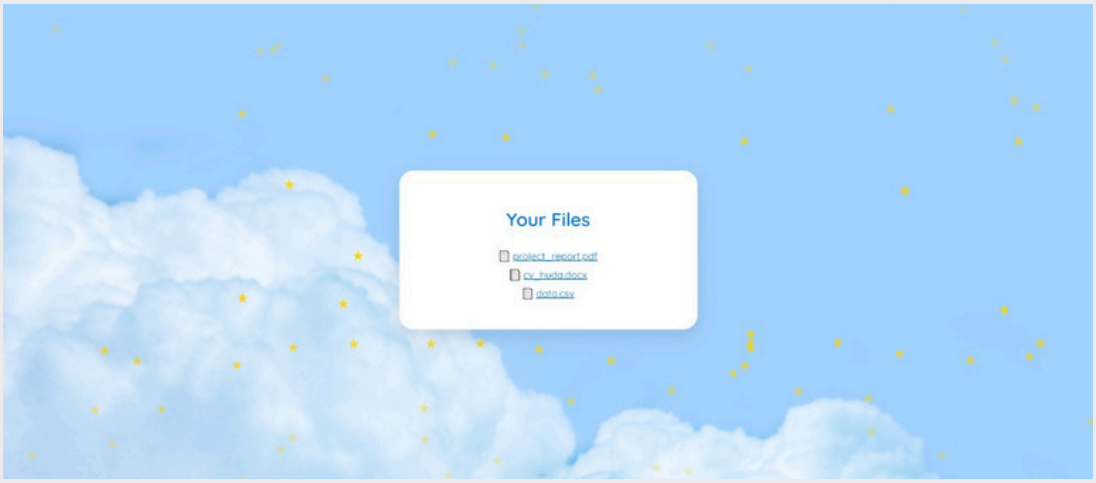
Step 5: Setting Up Networking (Amazon VPC)
We deployed our application within a custom VPC for isolation and control.
1. Creating the VPC:
o Created a VPC with a /16 CIDR block.
o Added a public subnet with a /24 block.
2. Internet Gateway and Route Table:
o Attached an internet gateway to the VPC.
o Configured a route table to allow traffic from the subnet to the internet.
3. Security Groups:
o Allowed HTTP (port 80) for web access.
o Allowed SSH (port 22) from our IP only.
These settings ensured the app was accessible online and secure from unwanted traffic.

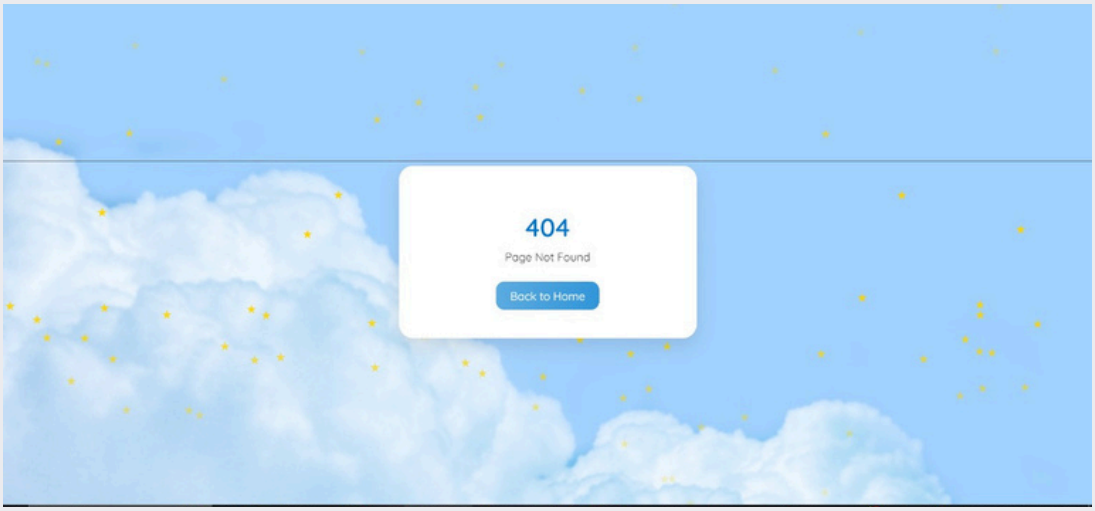
Step 6: Final Testing and Report Preparation

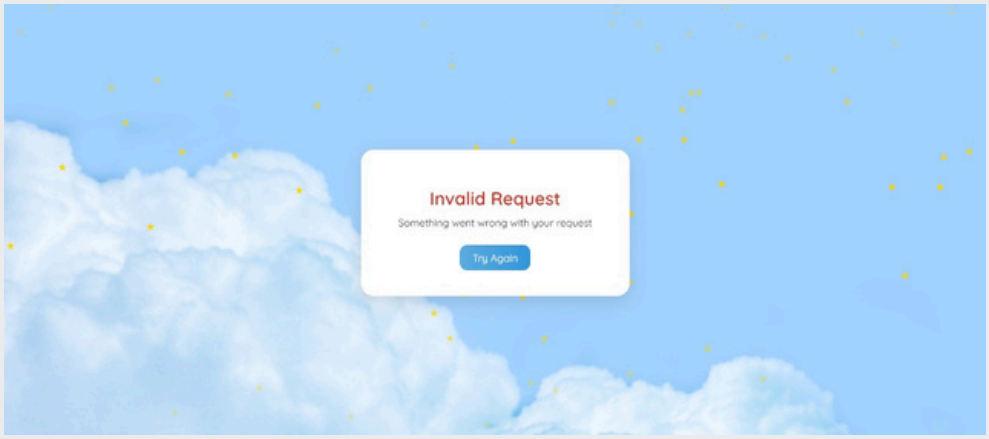
After deploying the application, we tested it thoroughly:
1. File Upload Testing:
o Uploaded various types of files: PDF, TXT, PNG, etc.
o Verified that invalid file types were rejected.
2. Download Link Verification:
o Ensured the generated download links worked correctly and pointed to S3.
3. Service Monitoring:
o Used systemctl to monitor the Node.js service.
o Restarted the service if any issue occurred.
Finally, we gathered all screenshots, verified every team member's contribution, and prepared this report.

Additional Information

File Limits: Each upload supports up to 10 files.
• Filename Handling: Filenames are replaced with unique 32-character strings to avoid conflicts.
• Security: Uploaded files are filtered to block any programming code or harmful scripts.
• Allowed File Types: txt, md, pdf, doc, docx, xls, ppt, pptx, jpg, jpeg, png, gif, bmp, webp, zip, rar, tar, gz.
• Backend Stack: Node.js with custom middleware for logging.
• Frontend Stack: HTML, CSS, JavaScript.
• Linux EC2 Commands: We used a custom shell script to install dependencies, clone the repository, and set up the server as a systemd service.
• SSH Access: Used to configure the instance and deploy the application.







Conclusion

Through this project, our team gained hands-on experience with deploying a full-stack web application on AWS. We learned how to configure and secure EC2 instances, use IAM roles effectively, manage file storage with S3, and handle file uploads in a scalable and efficient way. The project also helped us understand VPC configuration and the importance of proper permissions and monitoring. Overall, this was a valuable learning experience in cloud deployment, security, and teamwork.

We are proud of the results and look forward to building more advanced cloud-based systems in the future