



UNIVERSITY OF PETROLEUM AND ENERGY STUDIES
Dehradun-248007

School of Computer Science

2024

CLOUD PROJECT REPORT

4th SEMESTER

Submitted To: -
Mr. Abhirup Khanna

Submitted By: -
Group No: -6

S.No	Name	Batch	Sap ID	Roll NO.
1	Khushi Yadav	2	500119315	R2142231279
2	Priyambad Suman	1	500125300	R2142231260

1. INTRODUCTION

1.1 What is Cloud Computing?

Cloud computing is a technology that allows users to access and use computing resources—like servers, storage, databases, software, and analytics—over the internet, rather than owning and maintaining physical infrastructure. These resources are provided by cloud service providers (such as AWS, Microsoft Azure, or Google Cloud) and are available on-demand, scalable, and typically charged based on usage. It helps individuals and businesses reduce costs, improve performance, increase flexibility, and support remote access to services and data.

1.2 What is AWS?

AWS (Amazon Web Services) Launched in 2006 is a comprehensive and widely used cloud computing platform offered by Amazon. AWS helps businesses scale and grow efficiently by offering reliable, secure, and flexible cloud infrastructure. It provides a broad set of on-demand services such as computing power, storage, databases, networking, machine learning, and more, delivered over the internet on a pay-as-you-go basis. Popular services include Amazon EC2 (virtual servers), S3 (object storage), RDS (managed databases), and Lambda (serverless computing).

1.3 Purpose of Our Project:

The purpose of our project is to design and implement a Secure Document Management System (SDMS) using Amazon Web Services (AWS) , integrated with an existing Expenditure Management System (EMS). We chose this project to gain practical, hands-on experience in provisioning, configuring, and managing core AWS services such as EC2, S3, EBS, IAM, and Lambda—programmatically through the AWS SDK for Java. Given the rising demand for scalable cloud solutions in modern software systems, auto-scaling is a critical concept to learn and apply. By automating, the project prepares us for real-world cloud deployment scenarios.

2. Project Goals & Objectives

The goal is to provide a reliable, scalable, and secure way to handle and store documents such as bills, receipts, and financial reports that are generated within the EMS. This project uses various AWS services—EC2, S3, IAM, EBS, and Lambda—to automate backend processes, ensure secure access, and maintain high availability without manual intervention.

We use Amazon EC2 to host and run the Java-based EMS application, while Amazon S3 is used as the core storage service for storing and retrieving user-generated documents. To enhance the system's security and manage permissions correctly, we use IAM roles instead of hardcoding access credentials, allowing EC2 and Lambda functions to access services like S3 securely. AWS Lambda is integrated to automate tasks, such as deleting unused EC2 instances or processing uploaded documents, making the system smarter and cost-effective.

Additionally, Amazon EBS provides persistent disk storage to the EC2 instance, allowing logs or temporary documents to be stored securely even after rebooting. Overall, this project prepares us to work with cloud-native systems by demonstrating how to securely deploy, manage, and scale a real-world application using AWS services programmatically through the AWS SDK. It not only solves document security and storage challenges but also enhances the capabilities of the EMS by making it more automated, reliable, and production ready.

2.1 AWS Services Used and Their Purpose

1. Amazon EC2 (Elastic Compute Cloud)

- **Where Used:** Hosts the backend Java application (ExpenditureApp.jar) of EMS.
- **Purpose:**
 - Acts as a virtual server to run the application continuously.
 - Uses **user data scripts** to automatically install Java and launch the app at boot.
 - Provides a scalable compute environment without needing physical servers.

2. Amazon S3 (Simple Storage Service)

- **Where Used:** Stores reports, bills, invoices, and backend .jar file.
- **Purpose:**
 - Acts as the **central document storage system**.
 - Supports **file upload, download, versioning**, and lifecycle policies.
 - Used for storing the application JAR file which is downloaded by EC2 on startup.
 - Ensures **secure and scalable** storage of user and system-generated documents.

3. Amazon EBS (Elastic Block Store)

- **Where Used:** Attached to EC2 instance as additional persistent storage.
- **Purpose:**
 - Acts as a **virtual hard disk** for EC2.
 - Stores **temporary files, logs, or local backups** during app runtime.
 - Ensures that data remains intact even if EC2 restarts or crashes.

4. AWS IAM (Identity and Access Management)

- **Where Used:** IAM Roles are created and attached to EC2 and Lambda functions.
- **Purpose:**
 - **Grants secure access** to EC2 and Lambda without using hardcoded AWS keys.
 - Defines policies to allow only necessary permissions (e.g., S3 access, Lambda invoke).
 - Follows the **principle of least privilege** for secure and compliant architecture.

5. AWS Lambda

- **Where Used:** Automates tasks such as:
 - Deleting unused EC2 instances
 - Processing uploaded files
 - Managing backups or log cleanup
- **Purpose:**
 - Implements **event-driven automation** without needing a dedicated server.
 - Enhances system efficiency, reduces manual tasks, and lowers operating costs.

3. AWS Services Used & Their Configuration

AWS Region Used in All Services

Asia Pacific (Mumbai) — ap-south-1

3.1. IAM Roles and Permission Setup

• IAM Role for EC2

- **Role Name:** EC2_S3_Lambda_Role
- **Trust_Policy:**
Allows the EC2 service (ec2.amazonaws.com) to assume the role.
- **Attached AWS Managed Policies:**
 - AmazonS3FullAccess – Grants permission to read/write from S3 (for storing and retrieving reports and application files).
 - AWSLambda_FullAccess – Allows EC2 to trigger and interact with Lambda functions for automation.
- **Instance Profile:**
 - **Name:** EC2_S3_Lambda_Role_InstanceProfile
 - **Purpose:** Attached to EC2 instance to allow secure access to S3 and Lambda.

• IAM Role for Lambda Function

- **Role Name:** Lambda_EC2_Manage_Role
- **Trust_Policy:**
Allows the Lambda service (lambda.amazonaws.com) to assume the role.
- **Attached Policies:**
 - AWSLambdaBasicExecutionRole – Allows writing logs to Amazon CloudWatch.
 - AmazonEC2FullAccess – Grants Lambda permission to manage and terminate EC2 instances.

3.2. EC2 Instance Configuration

• Instance Launch Method

- Programmatically launched using AWS SDK for Java with specified AMI, key pair, IAM role, and user data script.

• AMI ID:

e.g., ami-0e35ddab05955cf57 (Amazon Linux or Ubuntu)

• Instance Type:

t2.micro (Free-tier eligible)

- **Key Pair Name:**

document-secure-key (used for SSH access and secure login)

- **Security Group:**

Allows access over HTTP (port 80), HTTPS (443), and SSH (22)

- **User Data Script (Base64 Encoded):**

- Installs Java
- Downloads the application JAR file from S3
- Launches the application automatically at boot

- **IAM Instance Profile:**

EC2_S3_Lambda_Role_InstanceProfile (used to access S3 and invoke Lambda)

3.3. S3 Bucket Configuration

Main Application Bucket

- **Bucket Name:** expenditure-docs-storage
- **Purpose:**
 - Stores the ExpenditureApp.jar file required for EC2
 - Stores reports, user documents, receipts, and generated files
- **Features Enabled:**
 - **Versioning:** Enabled to retain previous versions of files
 - **Encryption:** Server-side encryption (AES-256)
 - **Lifecycle Policy** (*optional*): For archiving old documents
- **Permissions Required:**
 - s3:CreateBucket
 - s3:PutObject
 - s3:GetObject
 - s3:ListBucket

3.4. EBS Volume Attachment

- **Device Name:** /dev/sdf
- **Volume Size:** 10 GiB
- **Volume Type:** gp2 (General Purpose SSD)
- **Attached To:** EC2 instance created via AWS SDK
- **Lifecycle:**
 - Volume is created during EC2 launch
 - Can be configured to delete on instance termination
- **Purpose:**

Stores runtime logs, local temp files, or document processing cache

3.5. AWS Lambda Function Configuration

- **Function Name:** AutoCleanupEC2Function
- **Region:** ap-south-1
- **Runtime:** Java 17
- **Handler:** Handler::handleRequest
- **IAM Execution Role:** Lambda_EC2_Manage_Role
- **Purpose:**
 - Automates the deletion of unused EC2 instances
 - Can be extended to handle scheduled S3 cleanups or PDF conversions
- **Source Code Location:**
 - Stored in S3 as a ZIP/JAR
 - Deployed from bucket: lambda-code-secure-bucket
- **Configuration Settings:**
 - **Timeout:** 30 seconds
 - **Memory:** 512 MB
 - **Triggers:** Can be event-based (e.g., via CloudWatch or manual)

4.Implementation Details & Code Snippets

4.1 Amazon S3

4.1.1 WHY DO WE NEED S3

We need a safe and scalable storage space to save our documents (like PDFs, reports, etc.) (here which is a .csv file) that we have generated from our Expenditure management, thus we use Amazon S3 (Simple Storage Service) for this purpose. We have also enabled versioning here. With the help of versioning, here we protect users from accidentally losing data and if someone overwrites your file, older versions are also not lost.

4.1.2 CODE LOGIC AND EXPLANATION

1.) First, we have Created a S3 Client: -

This establishes a connection to AWS S3. The reason for doing this is to enable the application to interact with S3, allowing it to store and retrieve files. Without this client, we wouldn't be able to access the S3 service for storing documents.

2.) Then we have Created a Bucket: -

A "bucket" is a storage container in S3 where files will be stored. This step ensures that you have a dedicated place in S3 to save your documents, such as financial reports, receipts, or other sensitive data. Creating a bucket is essential because S3 organizes data within these containers, and you need one to hold your files.

3.) After that we have Enabled Versioning: -

Enabling versioning in the S3 bucket allows you to keep track of multiple versions of a file. This is important because it provides a way to safely revert to previous versions if a file is accidentally overwritten or corrupted. In a document management system, versioning adds a layer of safety and helps in recovering earlier versions of documents, which is crucial when dealing with important files like financial records.

4.) Uploaded File:

Automatically uploads a document into the S3 bucket. It is essential because the core functionality of our system is to store documents securely in the cloud.

4.1.3 CODE SNIPPET

```
public class S3Manager {
    public static void createBucketAndUploadFile(String cloud-ems-project, String filePath, String keyName) {
        S3Client s3 = S3Client.builder()
            .region(Region.ap-south-1)
            .build();

        // Create bucket
        CreateBucketRequest createBucketRequest = CreateBucketRequest.builder()
            .bucket(cloud-ems-project)
            .createBucketConfiguration(CreateBucketConfiguration.builder()
                .locationConstraint(BucketLocationConstraint.ap-south-1)
                .build())
            .build();

        s3.createBucket(createBucketRequest);
        System.out.println("S3 Bucket created: " + cloud-ems-project);

        // Enable versioning
        PutBucketVersioningRequest versioningRequest = PutBucketVersioningRequest.builder()
            .bucket(cloud-ems-project)
            .versioningConfiguration(VersioningConfiguration.builder()
                .status(BucketVersioningStatus.ENABLED)
                .build())
            .build();

        s3.putBucketVersioning(versioningRequest);
        System.out.println("Versioning enabled on bucket: " + cloud-ems-project);
    }
}
```

Files and folders (1 total, 50.0 B)						
<div>Find by name</div>						
Name	Folder	Type	Size	Status	Error	
priyambad.csv	-	text/csv	50.0 B	Succeeded	-	

4.2 Amazon EC2 and EBS

4.2.1 WHY DO WE NEED EC2, EBS AND KEY PAIR

In our project, Amazon EC2 (Elastic Compute Cloud) plays the central role of hosting and running our Expenditure Management System. It acts as a virtual server where our Java application (ExpenditureApp.jar) for our Secure Document Management + Expenditure System is automatically deployed and executed. This ensures that users can interact with the system which is stable and is always available.

We attach **Amazon EBS (Elastic Block Store)** to the EC2 instance to provide **storage**. EBS acts like an external hard drive that retains data even if the instance stops or restarts. This is essential for securely storing application logs, user data, and files that may be used or generated by the system during runtime.

A **Key Pair** is required to securely access the EC2 instance using SSH (Secure Shell). It includes a public key stored on AWS and a private key downloaded by us. The key pair is attached during the EC2 instance creation, and it is crucial for administrative access, such as debugging, manual updates, or monitoring the system directly from the terminal. Without it, we cannot securely connect to the instance after it's launched.

4.2.2 CODE LOGIC AND EXPLANATION

This section of the project outlines the automated setup of a cloud server using AWS EC2 and EBS,

1. First we Create a EC2 Client:

This step initializes the **EC2 Client** using the AWS SDK for Java. [*The EC2 client is responsible for managing and interacting with EC2 services in AWS*], This allows the Java application to perform actions such as creating, managing, and interacting with EC2 instances.

Without creating this client, the Java application wouldn't be able to communicate with AWS services to launch EC2 instances or perform any other EC2-related operations.

How it is done:-

The `Ec2Client.create()` method is called to create a new instance of the EC2 client that can then be used to execute AWS EC2 API requests.

2. Define user Data:-

userData is a basically a script that is passed to the EC2 instance at it's launch. This script runs automatically when the EC2 instance starts. It installs Java and downloads the application from the S3 bucket, then runs the Expenditure Management GUI that we have made.

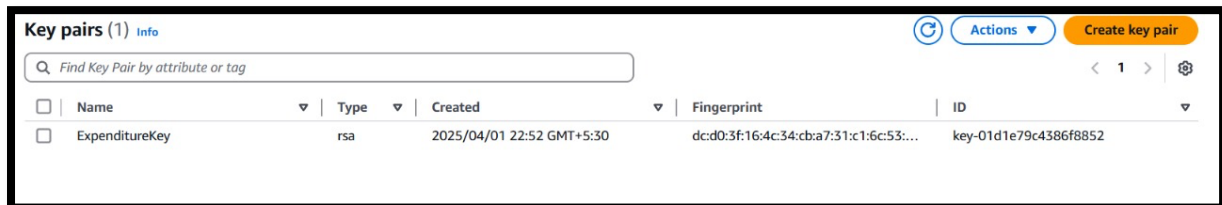
This script ensures that the EC2 instance is set up automatically with the necessary software and files when it starts, without manual intervention. It simplifies deployment and ensures consistency across all instances.

3. Specify Key Pair:

[Key pairs are essential for securely accessing your EC2 instance. Using SSH keys instead of passwords improves security and ensures only authorized users can log into the instance]

Here The **key pair** is used to enable and secure SSH access to the EC2 instance. When the instance is launched, a private key (ExpenditureKey) is used to securely connect to the instance

How it is done: The ExpenditureKey parameter is passed during the instance launch, specifying the key pair to use for secure SSH access.



4. Launching EC2 Instance:

This step launches a new EC2 instance using the provided Amazon Machine Image (AMI), instance type (e.g., t2.micro), security group (to control network access), IAM role (for permissions), user data script (to install Java and run the application), and key pair (for SSH access).

How it is done: The RunInstancesRequest is built with all the necessary parameters, and ec2.runInstances() is called to start the instance. The response includes information about the newly launched instance.

5. Then we have Created an EBS Volume:

An **Elastic Block Store (EBS) volume** is created, providing additional storage space for the EC2 instance. This volume is initially empty and can be used for storing data.

EC2 instances typically have limited storage, and EBS volumes provide persistent and scalable storage. This additional space can be used for various purposes, ensuring data is not lost when the EC2 instance is stopped or terminated.

How it is done: The CreateVolumeRequest is used to create a new volume of size 10 GiB .The volume is created in a specified availability zone (ap-south-1 in this case), and the volume type is set to GP2 (General Purpose SSD).

```

// Create and attach EBS volume
CreateVolumeRequest volumeRequest = CreateVolumeRequest.builder()
    .availabilityZone("ap-south-1")
    .size(10)
    .volumeType(VolumeType.GP2)
    .build();

CreateVolumeResponse volumeResponse = ec2.createVolume(volumeRequest);
String volumeId = volumeResponse.volumeId();
System.out.println("EBS Volume created with ID: " + volumeId);

// Wait for the volume to become available
DescribeVolumesRequest describeVolumesRequest = DescribeVolumesRequest.builder()
    .volumeIds(volumeId)
    .build();

boolean isAvailable = false;
while (!isAvailable) {
    DescribeVolumesResponse describeVolumesResponse = ec2.describeVolumes(describeVolumesRequest);
    VolumeState state = describeVolumesResponse.volumes().get(0).state();
    if (state == VolumeState.AVAILABLE) {
        isAvailable = true;
    } else {
        Thread.sleep(5000);
    }
}

// Attach the volume to the instance
AttachVolumeRequest attachRequest = AttachVolumeRequest.builder()
    .device("/dev/sdf")
    .instanceId(instanceId)
    .volumeId(volumeId)
    .build();

ec2.attachVolume(attachRequest);
System.out.println("EBS Volume attached to EC2 Instance.");

```

6. Wait Until Volume is Ready:

After creating the EBS volume, this step ensures that the volume reaches the **AVAILABLE** state before attempting to attach it to the EC2 instance. Volumes take some time to become available after creation. Trying to attach a volume before it's available would lead to errors. This step prevents such issues by ensuring the volume is ready for attachment.

How it is done: The DescribeVolumesRequest is called in a loop, and the script waits until the volume's state is "AVAILABLE." If the volume is not available, the loop pauses for 5 seconds before checking again.

7. Last Step is to Attach Volume:

After the EBS volume is available, it is attached to the EC2 instance at the device path.

How it is done: The AttachVolumeRequest is created, specifying the volume and instance to which it should be attached. The `ec2.attachVolume()` method is called to perform the attachment.

4.2.3 CODE SNIPPET

```
public class EC2Manager {
    public static void launchEC2Instance(String amiId, String instanceType, String keyName, String securityGroupId, String iamRoleName) {
        Ec2Client ec2 = Ec2Client.create();

        // User data script to install Java and run the application
        String userDataScript = "#!/bin/bash\n" +
            "yum update -y\n" +
            "yum install -y java-17-amazon-corretto\n" +
            "aws s3 cp s3://cloud-ems-project/ExpenditureApp.jar /home/ec2-user/\n" +
            "java -jar /home/ec2-user/ExpenditureApp.jar";

        RunInstancesRequest runRequest = RunInstancesRequest.builder()
            .imageId(amiId)
            .instanceType(instanceType)
            .keyName(keyName)
            .securityGroupIds(securityGroupId)
            .iamInstanceProfile(IamInstanceProfileSpecification.builder().name(iamRoleName).build())
            .userData(Base64.getEncoder().encodeToString(userDataScript.getBytes()))
            .minCount(1)
            .maxCount(1)
            .build();

        RunInstancesResponse response = ec2.runInstances(runRequest);
        String instanceId = response.instances().get(0).instanceId();
        System.out.println("EC2 Instance launched with ID: " + instanceId);
    }
}
```

Instances (1) Info

Last updated less than a minute ago

Connect

Instance state

Actions

Launch instances

All states

< 1 >

<input type="checkbox"/>	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4
<input type="checkbox"/>	Ems-cloud-pro...	i-09d7f02c153569c78	Running	t2.micro	Initializing	View alarms +	ap-south-1b	ec2-13-203

4.3 Lambda

4.3.1 WHY DO WE NEED LAMBDA FUNCTION

The goal of this AWS Lambda function is to automate the processing of expense files stored in Amazon S3, making the system efficient, smart, and cost-effective. Whenever a user uploads a CSV file containing expense data to an S3 bucket, this Lambda function is triggered automatically. It reads the file, calculates the total expenses, generates a formatted expense report, and then uploads that report back into the same S3 bucket. This eliminates the need for manual calculations and report generation by the user.

This setup ensures that every uploaded file is instantly processed, making the system real-time and self-managing. By using AWS Lambda, a serverless compute service, there is no need to manage or provision servers—you only pay for the compute time you use. This approach is significantly more cost-efficient than running EC2 instances constantly, and it helps your cloud infrastructure stay lightweight, automated, and secure.

In summary, the Lambda function serves as a backend automation tool that enhances your Secure Document Management System by allowing files to be processed securely, efficiently, and without manual effort—perfectly aligning with the goals of your Expenditure Management System.

Expenditure Manager - Dashboard (Priyambad)

Month: 05-May Budget for May: ₹10000.00 Set Budget

Add Expense All Expenses Statistics Generate Report Search & Filter Splitwise

Predefined Category: Food

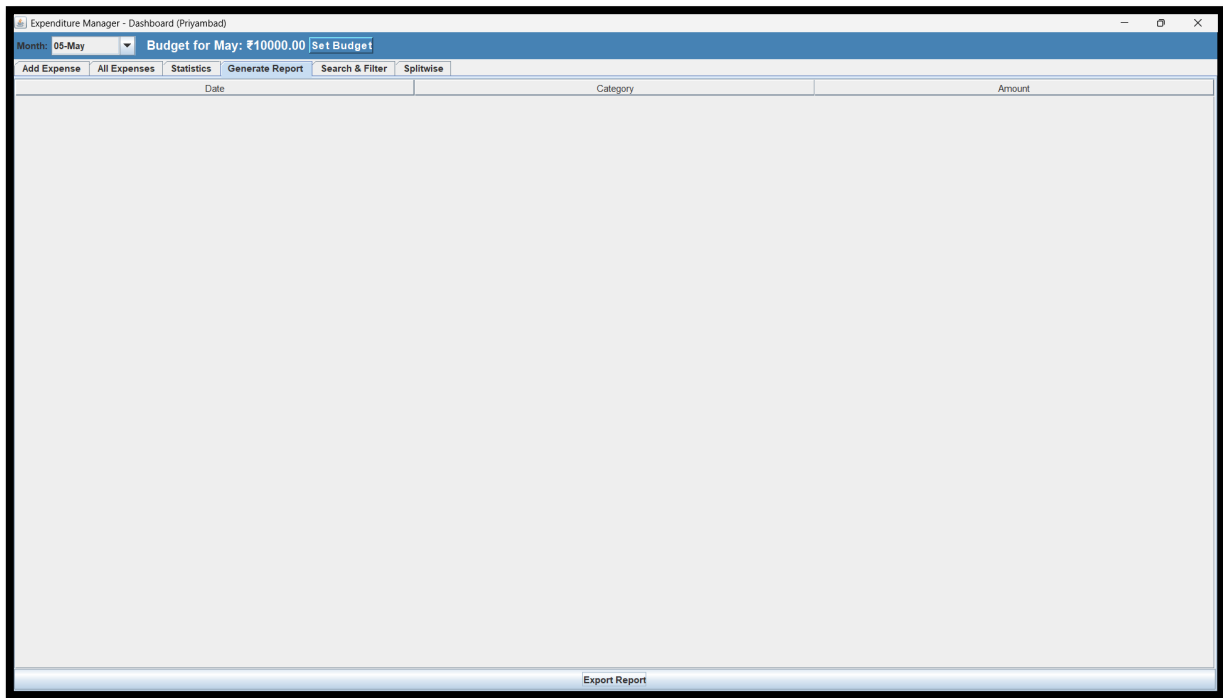
Or Custom Category:

Amount (₹):

Date (yyyy-MM-dd):

Description (max 100 characters):

Add Expense



[AWS Lambda is a serverless compute service that runs backend code in response to events—without the need to provision or manage servers]

4.3.1 CODE LOGIC AND EXPLANATION

1. **Connects to AWS Lambda using the AWS SDK in Java.**

This part of the code is responsible for automatically creating a Lambda function using Java and the AWS SDK. When we run this code, it first connects to AWS Lambda, allowing our application to interact with Lambda services.

2. **Reads your Lambda function's code from a .zip file.**

After That it reads the code for the Lambda function from a .zip file stored locally, which contains the logic we want AWS to run—like deleting unused EC2 instances.

3. **Creates the Lambda function with a name, role, runtime (Java17), and the code package.**

Next, it sends a request to AWS to create the Lambda function, specifying details like the function's name, the programming language runtime (Java 17), the IAM role for permissions, the handler method, and the function code itself.

4.3.2 CODE SNIPPET

```
public class ExpenseReportLambda implements RequestHandler<S3EventNotification, String> {
    public String handleRequest(S3EventNotification input, Context context) {
        // Get the S3 bucket and object key from the event
        S3EventNotification.S3EventNotificationRecord record = input.getRecords().get(0);
        String bucketName = record.getS3().getBucket().getName();
        String objectKey = record.getS3().getObject().getKey();

        // Download the expense file
        ResponseInputStream<GetObjectResponse> s3Object = s3Client.getObject(
            GetObjectRequest.builder()
                .bucket(bucketName)
                .key(objectKey)
                .build()
        );

        // Parse expenses
        BufferedReader reader = new BufferedReader(new InputStreamReader(s3Object));
        String line;
        double total = 0.0;
        StringBuilder report = new StringBuilder("Expense Report:\n\n");

        reader.readLine();
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split(",");
            if (parts.length == 3) {
                String date = parts[0];
                String category = parts[1];
                double amount = Double.parseDouble(parts[2]);
                total += amount;
                report.append(String.format("%s - %s: ₹%.2f\n", date, category, amount));
            }
        }


        report.append("\nTotal Expenses: ₹").append(String.format("%.2f", total));

        // Upload the report to S3
        String reportKey = "report-" + System.currentTimeMillis() + ".txt";
        s3Client.putObject(
```

Code properties [Info](#)

Package size
2.1 kB

SHA256 hash

 o0czLMg+EJlcQFujbmcsUZ7+4Omdgkex+rDhHid2knY=

Last modified

[52 seconds ago](#)

► Encryption with AWS KMS customer managed KMS key [Info](#)

Runtime settings [Info](#)

Runtime
Java 17

Handler [Info](#)

example.Hello::handleRequest

Architecture [Info](#)

x86_64

[Edit](#)

[Edit runtime management configuration](#)

► Runtime management configuration

4.4 IAM roles

4.4.1 WHY DO WE NEED IAM ROLES

IAM Roles (Identity and Access Management Roles) in AWS are a way to grant temporary access permissions to AWS services without using hardcoded credentials or access keys. Instead of attaching sensitive credentials directly to your application or EC2 instances, IAM roles provide a secure and manageable way to define who can access what resources and what actions they can perform.

In our Secure Document Management System integrated with the Expenditure Management System, IAM roles play a crucial role in enabling secure, automated, and permission-controlled access between services.

For example, we use an IAM role to allow our EC2 instance to access the S3 bucket—this is where our application JAR file, user documents, and reports are stored. Without the role, we would have to manually insert AWS keys into our code, which is unsafe and not recommended.

We also use another IAM role to allow our Lambda functions to manage EC2 instances, such as terminating them automatically when they're no longer needed. IAM roles make this possible without human intervention, ensuring both automation and security. These roles have helped us accomplish critical tasks like granting EC2 permission to download files from S3, invoking Lambda functions securely, and ensuring all interactions are logged and managed properly.

By using IAM roles, we've simplified permission management, followed AWS best practices for security, and enabled our project components to work together securely and efficiently without exposing sensitive information.

4.4.2 CODE LOGIC AND EXPLANATION

In this code, we are programmatically creating an IAM Role that allows our EC2 instance to securely access other AWS services like S3 and Lambda. Instead of hardcoding sensitive access keys into our application, we use this role to **grant** permissions the right way—by attaching predefined AWS policies such as `AmazonS3FullAccess` and `AWSLambda_FullAccess`.

This is done by first defining a trust policy that allows EC2 to assume the role, then creating the role using `CreateRoleRequest`, and finally attaching the required permissions using `AttachRolePolicyRequest`.

We do this because our EC2 instance needs to download the application JAR file from S3, and possibly trigger Lambda functions to automate tasks like cleaning up resources or processing files. By attaching this IAM Role to the EC2 instance at launch time, we ensure secure, managed, and scalable interaction with AWS services—exactly the kind of best practice AWS recommends for cloud-native systems.

```

public class IAMManager {
    public static void createIAMRole(String roleName) {
        IamClient iam = IamClient.create();

        String assumeRolePolicyDocument = "{\n" +
            "  \"Version\": \"2012-10-17\",\n" +
            "  \"Statement\": [\n" +
            "    {\n" +
            "      \"Effect\": \"Allow\",\n" +
            "      \"Principal\": {\n" +
            "        \"Service\": \"ec2.amazonaws.com\"\n" +
            "      },\n" +
            "      \"Action\": \"sts:AssumeRole\"\n" +
            "    }\n" +
            "  ]\n" +
            "}";

        CreateRoleRequest createRoleRequest = CreateRoleRequest.builder()
            .roleName(roleName)
            .assumeRolePolicyDocument(assumeRolePolicyDocument)
            .description("Role for EC2 with S3 and Lambda access")
            .build();

        iam.createRole(createRoleRequest);

        // Attach AmazonS3FullAccess policy
        AttachRolePolicyRequest attachPolicyRequest = AttachRolePolicyRequest.builder()
            .roleName(roleName)
            .policyArn("arn:aws:iam::aws:policy/AmazonS3FullAccess")
            .build();

        iam.attachRolePolicy(attachPolicyRequest);

        // Attach AWSLambdaFullAccess policy
        AttachRolePolicyRequest attachLambdaPolicyRequest = AttachRolePolicyRequest.builder()

```

1. Create IAM Client

First We create an IamClient object. We are doing this To interact with AWS IAM services using Java SDK.

How: IamClient iam = IamClient.create(); initializes the client needed to make IAM requests.

2. Define Trust Policy

Then A JSON string assumeRolePolicyDocument is created. This tells AWS **who can assume this role**—in this case, EC2 service.

How: "Principal": { "Service": "ec2.amazonaws.com" } allows EC2 to use this role.

3. Create IAM Role

After that A new IAM role is created with the defined trust policy. This is done To allow EC2 to access S3 and Lambda securely.

How: iam.createRole(createRoleRequest) sends the request to create the role.

4. Attach AmazonS3FullAccess Policy

Then it Attaches a managed AWS policy giving full access to S3. So that EC2 can **upload/download documents** from S3 buckets.

How: iam.attachRolePolicy(attachPolicyRequest) applies the S3 permission to the role.

5 **Attach AWSLambdaFullAccess Policy**

After that it Attaches AWSLambdaFullAccess policy to the same role. This lets EC2 (or other services) trigger and manage Lambda functions.

How: `iam.attachRolePolicy(attachLambdaPolicyRequest)` grants Lambda access to the role.

6 **Final Output**

Confirmation is printed to the console. This is To let us know the role and policies were successfully created.

How: `System.out.println(...)` provides a success message.

5.Challenges Faced & Solutions Implemented

5.1 Secure Access to AWS Services without Hardcoding Credentials

- **Problem:** We needed to access AWS S3 and Lambda from our EC2 instance without exposing AWS access keys.
- **Solution:** We created and attached an IAM Role to our EC2 instance using Java code. This gave EC2 temporary and secure permissions to access S3 and Lambda without hardcoded credentials.

5.2 Persistent Storage for Document Uploads

- **Problem:** EC2 instances have ephemeral storage which gets deleted after stop/terminate. We needed storage that persists.
- **Solution:** We created and attached an EBS volume to EC2 via Java code. This gave us persistent, block-level storage for documents. This Allowed safe and durable storage for uploaded and processed documents.

5.3 Automating File Processing Tasks

- **Problem:** Manual processing of uploaded files (like converting Word to PDF, resizing images) would be inefficient and error prone.
- **Solution:** We used AWS Lambda functions to automate tasks like converting Word documents to PDF and resizing images, triggered by user actions or S3 uploads. This Reduced manual effort, ensured fast processing, and allowed scaling automatically.

5.4 Creating Resources Programmatically

- **Problem:** Manually setting up AWS resources (IAM, EC2, EBS, Lambda) would be slow and inconsistent.
- **Solution:** We used AWS SDK for Java to write code that creates and configures all necessary services (e.g., EC2, IAM roles, S3 access).This Ensured consistency, reusability, and faster setup for future deployments.

5.5 Secure Login and User Data Management

- **Problem:** Ensuring secure user login and management for document access within our expenditure system.
- **Solution:** We integrated authentication in our Java Swing GUI, and managed document access through controlled S3 permissions and Lambda triggers. This Protected user data and documents with role-based access and AWS IAM integration.

6. Future Enhancements

In the future, our Secure Document Management System can be enhanced with advanced features like Optical Character Recognition (OCR) for reading text from images and scanned documents, and AI-based document classification using Amazon Comprehend or SageMaker.

We can also implement version control for documents stored in S3, enabling users to track changes and restore older versions. Integration with Amazon Rekognition could provide image moderation and content analysis.

Additionally, we can also implement multi-user access control with fine-grained permission and audit logging using AWS CloudTrail will further strengthen security.

We can also use Amazon CloudFront for fast content delivery and Auto Scaling Groups for EC2 will enhance performance and scalability as user load grows.