CNG-495 Capstone Project Stage 3: Final Report of E-Commerce Application with ML Capabilities

Cem KARASU 2315398

Metehan YILMAZ 2315638

11/16/23

Table of Contents

Project Topic

Developing a Flutter-based e-commerce application on the AWS cloud infrastructure with integrated machine learning capabilities.

Project Description

This project aims to create an e-commerce application where users can conveniently list their products by adding new items. AWS Cognito manages the authentications of each user and customer. Customers can search and add products to their cart. Payment microservice is not included in this project.

This application solves the problem which is faced during listing. It can be listed wrongly categorized due to human error. Our application eliminates this problem by automatically choosing the category for the product based on its image.

To achieve this, the application utilizes image recognition technology from Amazon Rekognition. When users upload an image to add a new item, Amazon Rekognition automatically identifies the object and selects the category for the product. This functionality streamlines the process of filling in item details by automatically populating certain fields based on previously configured specifications of the identified object.

Cloud Delivery Models

In our project, we plan to adopt serverless computing as a fundamental part of our architecture. We aim to leverage AWS Lambda for its remarkable ability to automatically scale and handle event-driven tasks. Our focus will be on the simplicity and scalability provided by Lambda and AWS Cognito for serverless user management. We will provide software developed in flutter.

- **SaaS (Software as a Service):** We will provide an e-commerce application that users can list products.

Architectural Overview

User Authentication:

AWS Cognito for serverless user management, ensuring secure and scalable authentication.

Static and Dynamic Content Delivery:

CloudFront for routing static requests to S3 bucket holding static data (e.g., images).

API Gateway as a secure single point of access for dynamic requests, routing to microservices.

Microservices:

Four microservices - Inventory, Cart, Order, and Search - implemented using AWS Lambda.

DynamoDB for data storage used by Inventory, Cart, and Order microservices.

Workflows:

Three workflows: New Order Workflow, Cancel Order Workflow (handled by Order MicroService), and Image Recognition Workflow (handled by Inventory MicroService using Lambda Step Functions).

Image Recognition Workflow:

Triggered when an image is uploaded to S3.

Inventory MicroService Step Function invokes Amazon Rekognition.

Recognized labels stored in DynamoDB, enhancing product information.

Data Storage and Processing:

DynamoDB used by Inventory, Cart, and Order microservices.

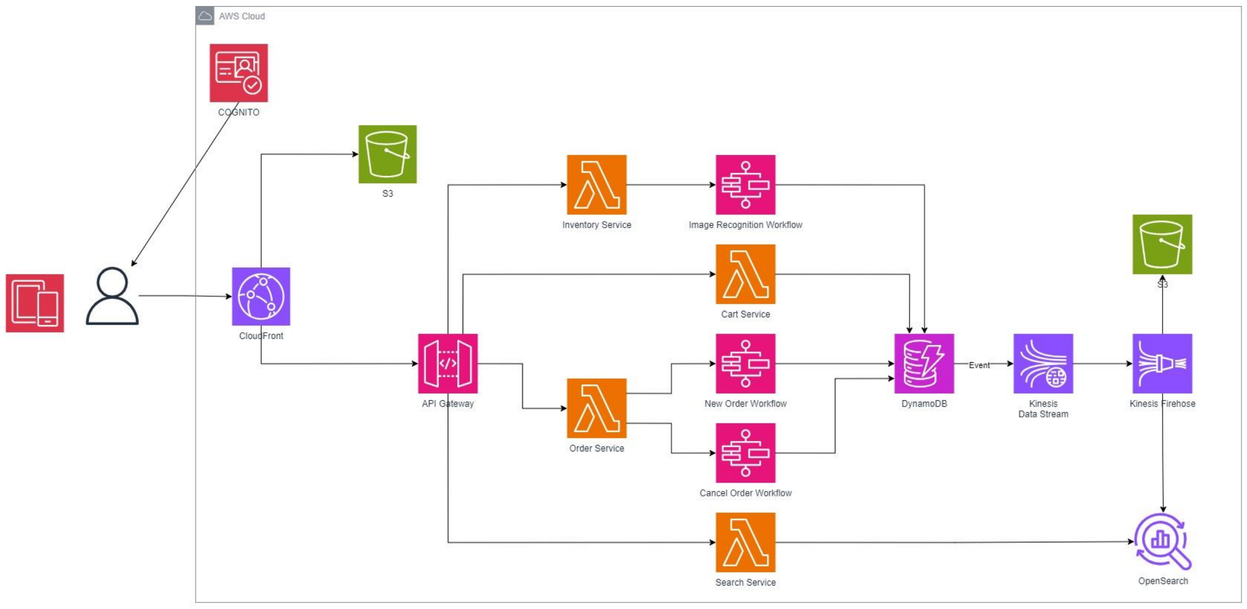
Kinesis Data Stream for data streaming.

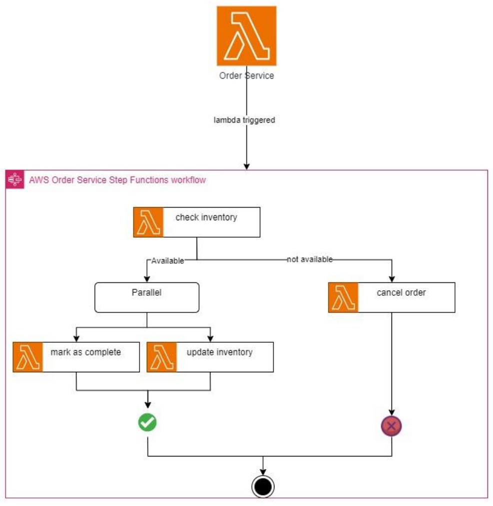
Kinesis Firehose for efficient data delivery to S3 Bucket and OpenSearch.

Search MicroService:

Utilizes OpenSearch for efficient and scalable search functionality.

Diagrams/Figures





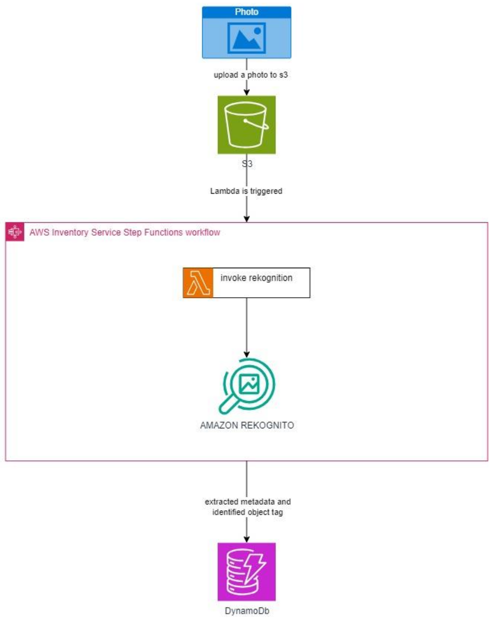


Photo
upload a photo to s3
S3
Lambda is triggered
AWS Inventory Service Step Functions workflow
invoke rekognition
AMAZON REKOGNITO
extracted metadata and identified object tag
DynamoDb

Expected Contribution

We've outlined how each team member will contribute to our project, considering our individual strengths and areas of expertise:

Front-End Flutter Mobile Application Development:

Cem will be responsible for developing 100% of the front-end Flutter mobile application. This includes creating the look and feel of the app to ensure a smooth and easy shopping experience.

Report Writing:

Metehan will handle most of the report writing, covering 90% of this task. Cem will contribute 10% to the report writing. We'll keep communication open throughout to make sure the report is well-rounded.

Cloud Computing Tasks:

To make sure the workload is balanced, we're dividing cloud computing tasks into 60% and 40%. Metehan will take on 60%, and Cem will manage the remaining 40%. Both of us will work closely on all cloud computing tasks to understand and implement them effectively.

We understand the importance of staying in touch and working together closely. This plan is designed to make the most of each team member's skills, ensuring our platform on the AWS cloud comes together successfully.

References

AWS Lambda vs. EC2: Which is Right for Your Use Case?

AWS Lambda Pricing

AWS Step Function Pricing

Amazon Cognito Pricing

Amazon CloudFront Pricing

Amazon S3 Pricing

Amazon API Gateway Pricing

Amazon SWF (Simple Work Flow) Pricing

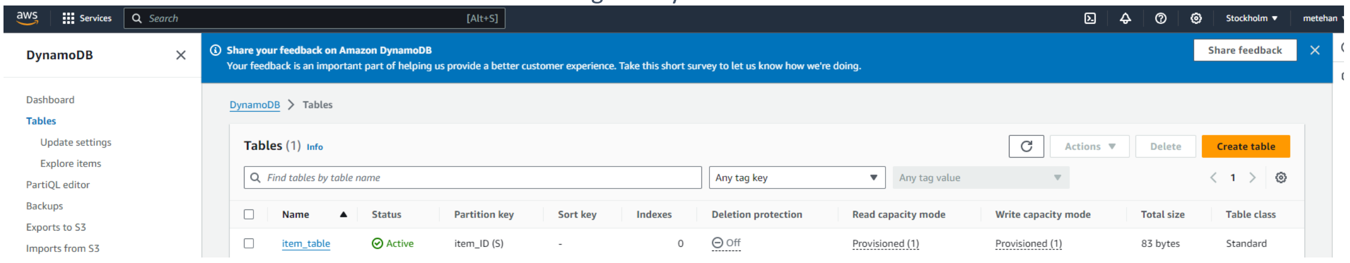
Amazon DynamoDB Pricing

Stage 2: Progress Report

Milestones Achieved:

Milestone Name	Description	Week	Member
Proposal	Proposal for this project is prepared to be submitted.	W7	All
Self-Study	Each group members get ready for the continuation of the project by doing self-study on the AWS topics.	W9	All
API Gateway	Triggers for the Lambda Functions and S3 bucket are created.	W10	Metehan
Lambda Functions	"items" and "inventory" Lambda Functions are implemented using API Gateway endpoints. More specifically search, add, and delete functionalities are implemented.	W10	Cem
DynamoDB	Simple item_table is created to be used by Lambda Functions to store data to DynamoDB.	W10	Cem
GitHub Commits	GitHub commits are done by a group member even though codes are written by either of the group member.	W10	All
Progress Report	Stage 2 of the capstone project.	W11	All

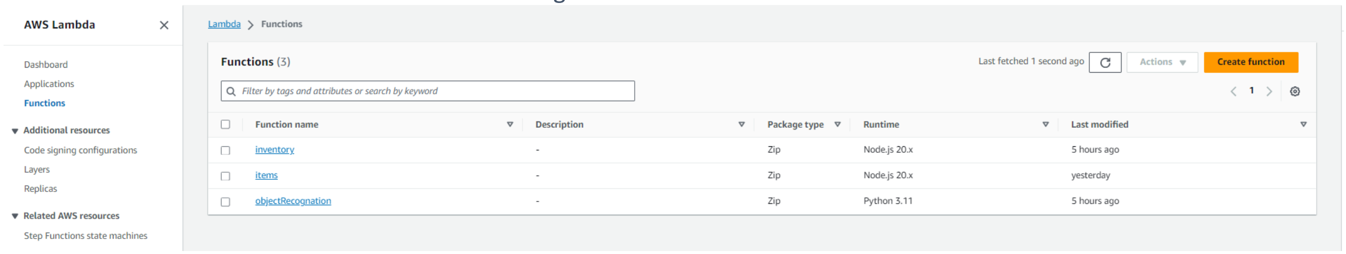
Figure: DynamoDB



Share your feedback on Amazon DynamoDB
Your feedback is an important part of helping us provide a better customer experience. Take this short survey to let us know how we're doing.

Name	Status	Partition key	Sort key	Indexes	Deletion protection	Read capacity mode	Write capacity mode	Total size	Table class
item_table	Active	item_ID (S)	-	0	Off	Provisioned (1)	Provisioned (1)	83 bytes	Standard

Figure: Lambda Functions



Functions (3)

Function name	Description	Package type	Runtime	Last modified
inventory	-	Zip	Node.js 20.x	5 hours ago
items	-	Zip	Node.js 20.x	yesterday
objectRecognation	-	Zip	Python 3.11	5 hours ago

https://github.com/metumetehan/e-commerce-CNG-495.git

Milestones Remained:

Milestone Name	Description	Week	Member
UI	The front-end of the project has not been completely implemented. Therefore, it is not included in this report.	W11	Cem
Image Recognition	Implementing Amazon Rekognito.	W11	Metehan
Integration of UI	Amazon Rekognito and UI integration. Add functionality to suggest and autofill information while adding an item. Uploading an image triggers Rekognito and object recognition information is retrieved to the UI.	W12	All

List of Futures to be delivered:

- Server code.
- Database backup.
- https://github.com/metumetehan/e-commerce-CNG-495.git

Stage 3:

E-commerce Application with ML Capabilities

This project aims to create an e-commerce application where users can conveniently list their products by adding new items. AWS Cognito manages the authentications of each user and customer. Customers can search and add products to their cart. Payment microservice is not included in this project.

This application solves the problem which is faced during listing. It can be listed wrongly categorized due to human error. Our application eliminates this problem by automatically choosing the category for the product based on its image.

To achieve this, the application utilizes image recognition technology from Amazon Rekognition. When users upload an image to add a new item, Amazon Rekognition automatically identifies the object and selects the category for the product. This functionality streamlines the process of filling in item details by automatically populating certain fields based on previously configured specifications of the identified object.

Trendyol is an e-commerce application that is like our application. Main differences are we use image recognition for deciding a category automatically while Trendyol has more capabilities such as payment.

GitHub repository for the back end: https://github.com/ChesterMETU/e_commerce_CNG495.git

GitHub repository for the front-end UI: https://github.com/ChesterMETU/e_commerce_CNG495_ui.git

And you can find the old repository here: https://github.com/metumetehan/e-commerce-CNG-495.git

Authentication

For authentication, we use AWS Amplify with AWS Cognito. Using Amplify Studio in AWS Console, first we created an Amplify application. Then, to setup authentication, we created a login mechanism and configured it to be logged in using email and password. and we created a sign-up mechanism and configured it to be registered using username, email and password. Then, for the front-end mobile application, we use three Flutter modules: amplify_flutter, amplify_auth_cognito, and amplify_authenticator. Using AWS command line interface, Amplify configurations are pulled using "amplify pull" command. Amplify Authenticator module is used for the UI of login and register.

Managing Items

Adding An Item

To add an item, we created a lambda function that is triggered by making a *PUT* request to the following API endpoint: https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/items

In the request body, *name, price, piece, creater_id, product_description*, and *file* must be included. Here is an example body of a request:

```
    "name": nameController.value.text,
    "price": int.parse(priceController.value.text),
    "piece": int.parse(pieceController.value.text),
    "creator_id": user_id,
    "product_description": descriptionController.text,
    "file": base64img
```

After the Lambda function is triggered, first we create a new item to the DynamoDB table called *item_table* with the attributes in the request body without image *file* and its category is initially set to unknown. After that, item image provided in the request body as base64 encoded image is decoded and put to S3 Bucket called *rekonimage* by setting key to the *item_id*. You can see the associated function below.

```
case "PUT /items":
        let requestJSON = JSON.parse(event.body);


        var IDcounter =  await dynamo.send(
          new GetCommand({
            TableName: counterTable,
            Key: {
              counter: "1",
            },
          })
        );

        IDcounter = IDcounter.Item.counterID;

        await dynamo.send(
          new UpdateCommand({
            TableName: counterTable,
            Key: {
              counter: "1"
            },
            UpdateExpression: 'set #c = :x',
            ExpressionAttributeNames: {'#c' : 'counterID'},
            ExpressionAttributeValues: {
```

```
            ':x' : IDcounter + 1,
          }
        })
      );

      await dynamo.send(
        new PutCommand({
          TableName: tableName,
          Item: {
            item_ID: `${IDcounter}`,
            price: requestJSON.price,
            piece: requestJSON.piece,
            name: requestJSON.name,
            creator_id: requestJSON.creator_id,
            image: requestJSON.file,
            description: requestJSON.product_description,
            category: "unknown"
          },
        })
      );


      const base64Image = requestJSON.file;
      const decodedImage = Buffer.from(base64Image,"base64");
      const command = new PutObjectCommand({
        Bucket: "rekonimage",
        Key: `${IDcounter}`,
        Body: decodedImage,
        ContentType: "image/jpeg"
      });
      const response = await S3.send(command);
      body = requestJSON.file;
      break;
```

Getting All Items

To get all items from the database, we created a lambda function that is triggered by making a *GET* request to the following API endpoint: https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/items

After the Lambda function is triggered, first we send a dynamo scan command to the DynamoDB table called *item_table* with only the table name that is *item_table*. And it returns the list of all the items in the database. Then, it is put in the response body and sent to the client. You can see the associated function below.

```
      case "GET /items":
        body = await dynamo.send(
          new ScanCommand({ TableName: tableName })
        );
        body = body.Items;
        break;
```

Getting An Item by ID

To get an item from the database, we created a lambda function that is triggered by making a *GET* request to the following API endpoint: https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/items/{id}

Item id must be placed as a path parameter. Here is an example:

https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/items/5

After the Lambda function is triggered, first we send a dynamo *scan* command to the DynamoDB table called *item_table* with the table name and key parameter which is the primary key of the *item_table* and it is taken from path parameter in the request URL. And it returns the list of all the items in the database. Then, it is put in the response body and sent to the client. You can see the associated function below.

```
      case "GET /items/{id}":
        body = await dynamo.send(
          new GetCommand({
            TableName: tableName,
            Key: {
              item_ID: event.pathParameters.id,
            },
          })
        );
        body = body.Item;
        break;
```

Deleting An Item by ID

To delete an item from the database, we created a lambda function that is triggered by making a *DELETE* request to the following API endpoint: https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/items/{id}

Item id must be placed as a path parameter. Here is an example:

https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/items/5

After the Lambda function is triggered, first we send a dynamo *delete* command to the DynamoDB table called *item_table* with the table name and key parameter which is the primary key of the *item_table* and it is taken from path parameter in the request URL. And it returns the list of all the items in the database. Then, it is put in the response body and sent to the client. You can see the associated function below.

```
      case "DELETE /items/{id}":
        await dynamo.send(
          new DeleteCommand({
            TableName: tableName,
            Key: {
              item_ID: event.pathParameters.id,
            },
          })
        );
        body = `Deleted item ${event.pathParameters.id}`;
        break;
```

Getting Item Categories

To get all the item categories from the database, we created a lambda function that is triggered by making a *GET* request to the following API endpoint: https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/itemcategorys

After the Lambda function is triggered, first we send a dynamo *scan* command to the DynamoDB table called *item_table* with the table name. And it returns the list of all the items in the database. Then, all the item categories in the list are put into a dictionary and distinct categories are send to the client via the response body. Here is the associated function:

```
      case "GET /itemcategorys":
        var items = await dynamo.send(
          new ScanCommand({ TableName: tableName })
        );
        items = items.Items;
        const distinctValues = {};

        items.forEach(item => {
          const attributeValue = item["category"];
          distinctValues[attributeValue] = true;
        });

        body = Object.keys(distinctValues);
        break;
```

Image Rekognition

To decide the item category of an item, we created a lambda function that is triggered whenever an item is put into the S3 Bucket called *rekonimage*. First, the function gets bucket name and key of the item from the event header. Using those parameters, the function gets the image from S3 Bucket and then it is sent to AWS Rekognition. AWS Rekognition returns the data that we can get the item category from. Then, the function updates the item category in the database by sending *update* command with the table name which is *item_table* and key which is S3 item key.

```
bucket = event['Records'][0]['s3']['bucket']['name']
key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'],
encoding='utf-8')
print(bucket,key);
try:
    result = rk.detect_labels(
    Image={'S3Object':{'Bucket':bucket,'Name': key},},
        MaxLabels=1,
        Features=[
    'GENERAL_LABELS',
    ],
        )
    print(result)


    update_response = db.update_item(
        TableName= "item_table",
        Key={"item_ID": {
            "S": key
            }
        },
        ExpressionAttributeNames={
            '#C': 'category',
        },
        ExpressionAttributeValues={
            ':c': {
                "S": result["Labels"][0]["Categories"][0]["Name"],
            },
        },
        UpdateExpression='SET #C = :c',
        ReturnValues="UPDATED_NEW",
    )


    return "Success"
except Exception as e:
    print(e)
    print('Error getting object {} from bucket {}. Make sure they exist and your bucket is in the same region as this function.'.format(key, bucket))
    raise e
```

Inventory

To asd, we created a lambda function that is triggered whenever an

Part	**AWS Service**	**Programming Language**	**Lambda Function Name**
Authentication	AWS Cognito & Amplify	Node.js 16.x Node.js 18.x	amplify-login-create-auth-challenge-56c36a45 amplify-login-custom-message-56c36a45 amplify-ecommerceflutter--UpdateRolesWithIDPFuncti-TeqqfJtbBnas amplify-login-verify-auth-challenge-56c36a45 amplify-login-define-auth-challenge-56c36a45
Managing Items	DynamoDB & S3 Bucket	Node.js 20.x	items
Image Recognition	AWS Rekognito & S3 Bucket	Python 3.12	rekonObject