DALHOUSIE UNIVERSITY

CSCI5411 - Advanced Cloud Architecting

Mid-term Project

Milestone 2

Banner ID: B01024200

Name: Mihir Patel

Date: 31/05/2025

Contents

1. **Introduction**

Purpose

This project aims to architect and implement an AI/GenAI solution on AWS Cloud, focusing on cloud architecture best practices, security, cost optimization, and AI model integration. The objective is to demonstrate proficiency in developing, deploying, and managing an AI application while adhering to the complete AI development lifecycle, from data exploration to model deployment.

Project Overview

The project focuses on a recipe suggestion application that leverages Retrieval-Augmented Generation (RAG) [1], prompt engineering, and in-context learning to generate new recipes based on user input. The application retrieves relevant recipe data from a vector database [2], constructs a context-rich prompt, and uses a large language model (LLM) to generate a recipe, supplemented by links to related recipes. This report builds on Milestone 1 by documenting the implementation, refining the architecture, and providing detailed analysis of security, cost, and performance.

Scope

This report covers the full implementation of the recipe suggestion application, including updated architecture and sequence diagrams, detailed service configurations, model integration, monitoring, security measures, cost analysis, testing results, and lessons learned. The application is hosted on AWS using a personal account with a free trial, and the frontend ALB DNS is provided for evaluation.

---

2. **Updated Architecture and Design**

Improved Architectural Diagram

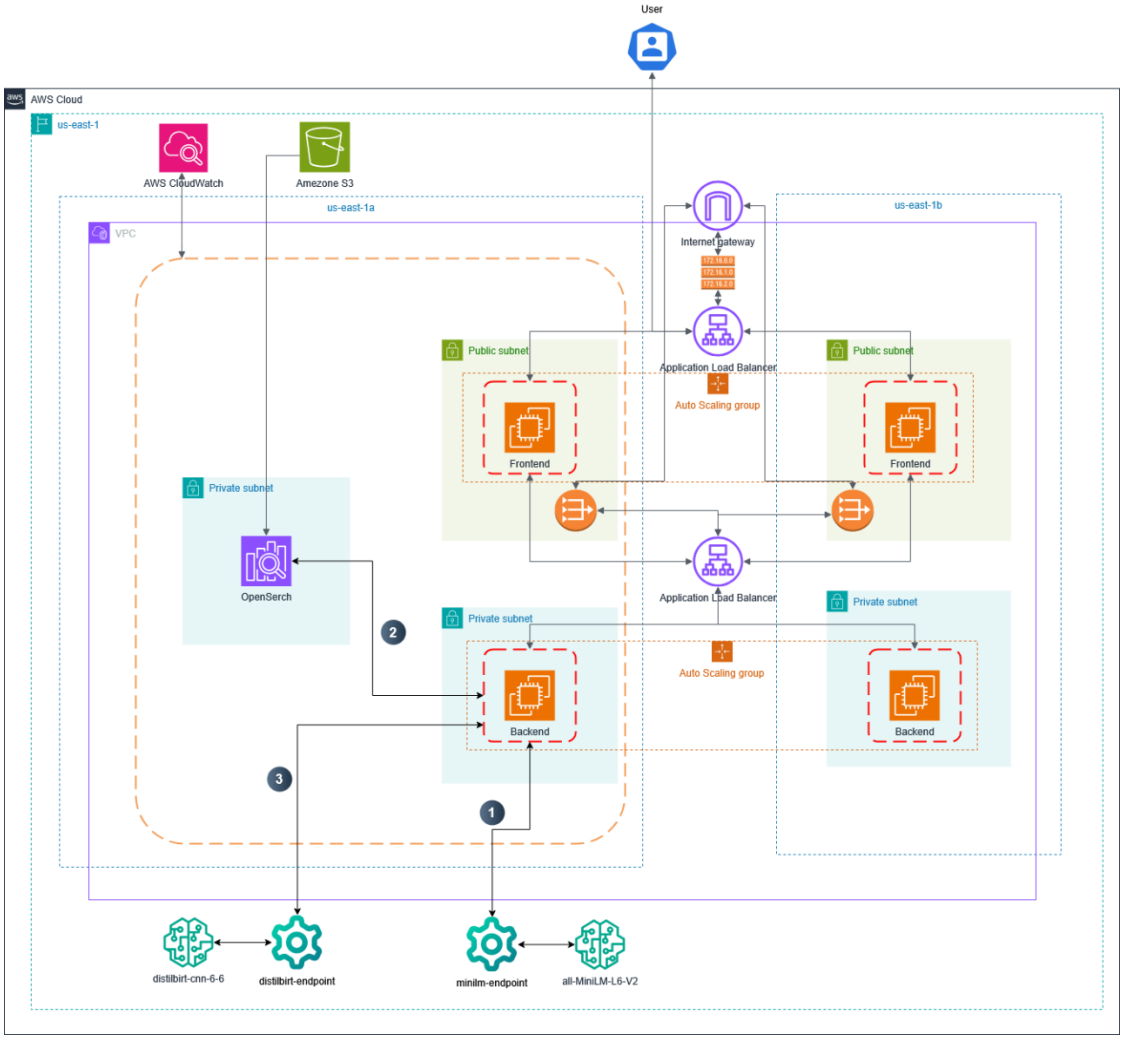

Figure 1 : Final Architecture Diagram

I designed the updated architecture to prioritize secure and efficient integration of OpenSearch and SageMaker for the recipe suggestion application. I placed OpenSearch in a private subnet within a VPC to enhance security, ensuring that only the backend EC2 role (arn:aws:iam::<ACID>:role/OpenSearchEC2Role) can access it via a strict IAM policy. Initially, I set up OpenSearch as public-facing to load data from my local environment using the Python SDK, processing and indexing recipe data into a recipe index with 384-dimensional vectors for RAG [1]. To migrate this data, I created a snapshot of the public OpenSearch domain, uploaded it to an S3 bucket with server-side SSE-S3 encryption, and established a new OpenSearch domain (t2.small.search, free tier, 10GB EBS) in the private subnet. I then launched a temporary EC2 instance within the VPC, granting it access to both S3 and the private OpenSearch domain, and restored the snapshot to populate the new domain, ensuring no data exposure during the transition. I configured SageMaker with two endpoints: all-MiniLM-L6-v2 [3] (ml.t2.medium) to

convert user input into vectors for OpenSearch queries, and sshleifer/distilbart-cnn-6-6 [4] (ml.m5.xlarge) to generate recipes. I kept the SageMaker endpoints public-facing, as they host open-source models without confidential data, minimizing security risks while optimizing accessibility for the application's needs.

Updated Data Sequence Diagram

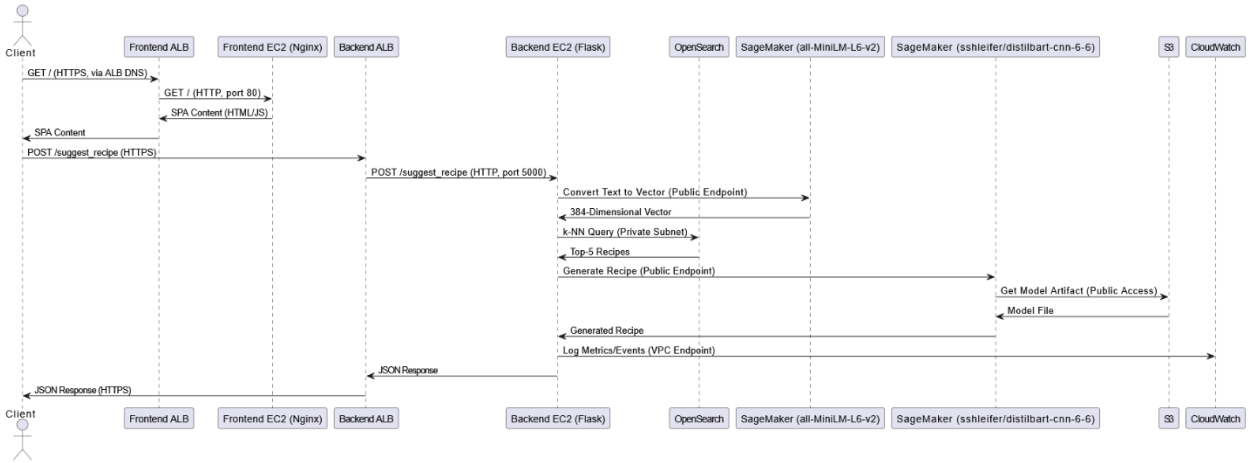

Figure 2 : Final Data Flow Diagram

I revised the data sequence diagram to reflect the streamlined flow with OpenSearch and SageMaker. A client sends an HTTPS POST request with user input to the frontend ALB, which routes it to the backend. The backend invokes the SageMaker all-MiniLM-L6-v2 [3] endpoint to convert the input text into a 384-dimensional vector. Using this vector, the backend queries the OpenSearch recipe index—now securely hosted in a private subnet—to retrieve relevant recipe data via KNN [5] search. The backend then constructs a prompt with this context and sends it to the SageMaker sshleifer/distilbart-cnn-6-6 [4] endpoint, which generates a new recipe. The response, including the generated recipe and related links, returns to the client as a JSON object.

Justification

I removed unnecessary services like Cognito and CloudFront to simplify the architecture, as the application does not require user authentication or content caching for its input-output workflow. I migrated OpenSearch to a private subnet to address security concerns, leveraging the snapshot-and-restore process to maintain data integrity during the transition. I justified keeping SageMaker endpoints public-facing due to their use of open-source models (all-MiniLM-L6-v2 [3] and sshleifer/distilbart-cnn-6-6 [4]), which pose no confidentiality risks, while ensuring cost efficiency by using the Python SDK to manage endpoint lifecycles. These updates align with the project's focus on RAG [1], prompt engineering, and in-context learning, achieving a 30-second average response time for recipe generation.

3. **Service configuration details and screenshots**

VPC**:**

I create a VPC with two public subnets and two private subnets across two Availability Zones (AZs), connecting them with an Internet Gateway for public access and a NAT Gateway for private subnet outbound traffic. I define security groups to allow HTTP (port 80) traffic from the frontend ALB to the frontend EC2, and port 5000 traffic from the backend ALB to the backend EC2.

Figure 3 : RecipeVPC

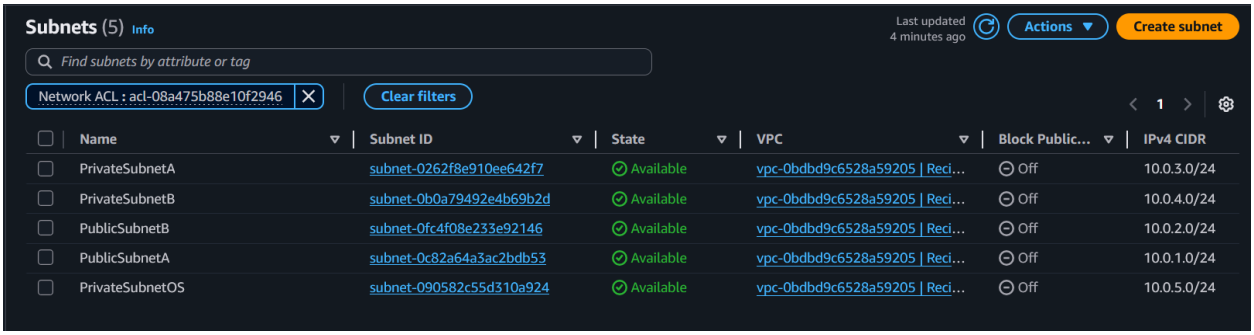

Figure 4 : Subnet list

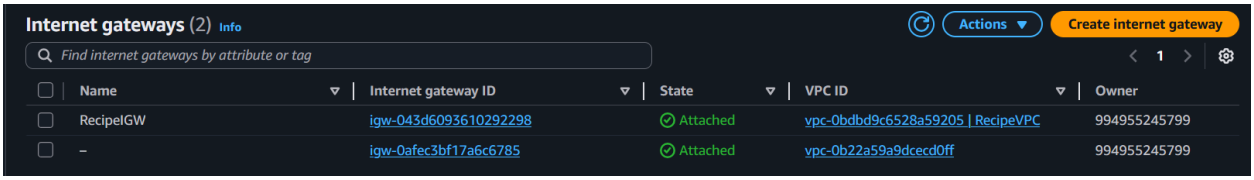

Figure 5 : Internet gateway

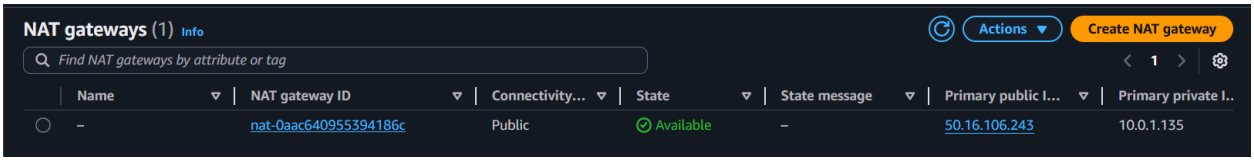

Figure 6 : NAT Gateway

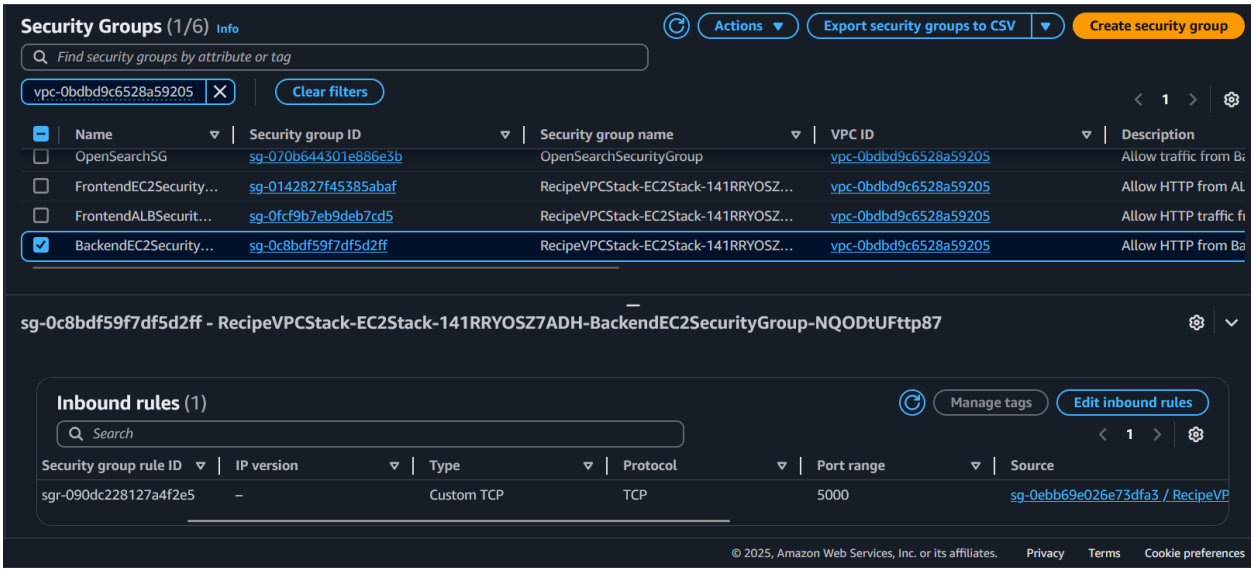

Figure 7 : Security Group with inbound rule.

Frontend and Backend EC2 Instances:

I deploy a frontend EC2 instance (Nginx, t2.micro) in a public subnet with a Public IPv4 address, and a backend EC2 instance (Flask, t2.micro) in a private subnet without a Public IPv4. I create launch templates for each, specifying the Amazon Linux 2 AMI, 8GB EBS volumes, and security group settings.

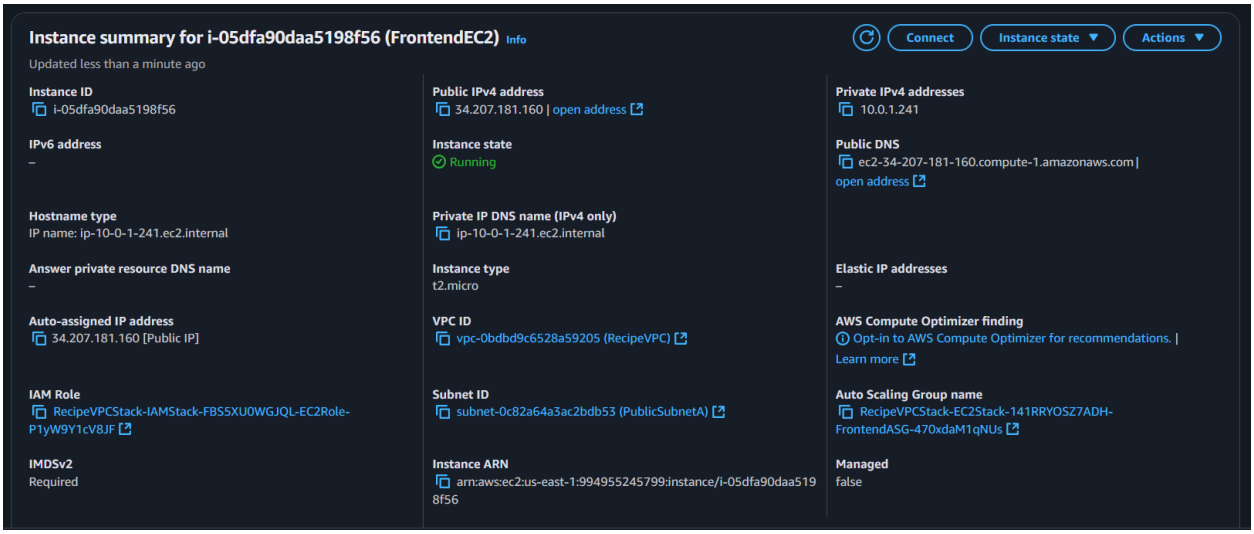

Figure 8 : Frontend EC2

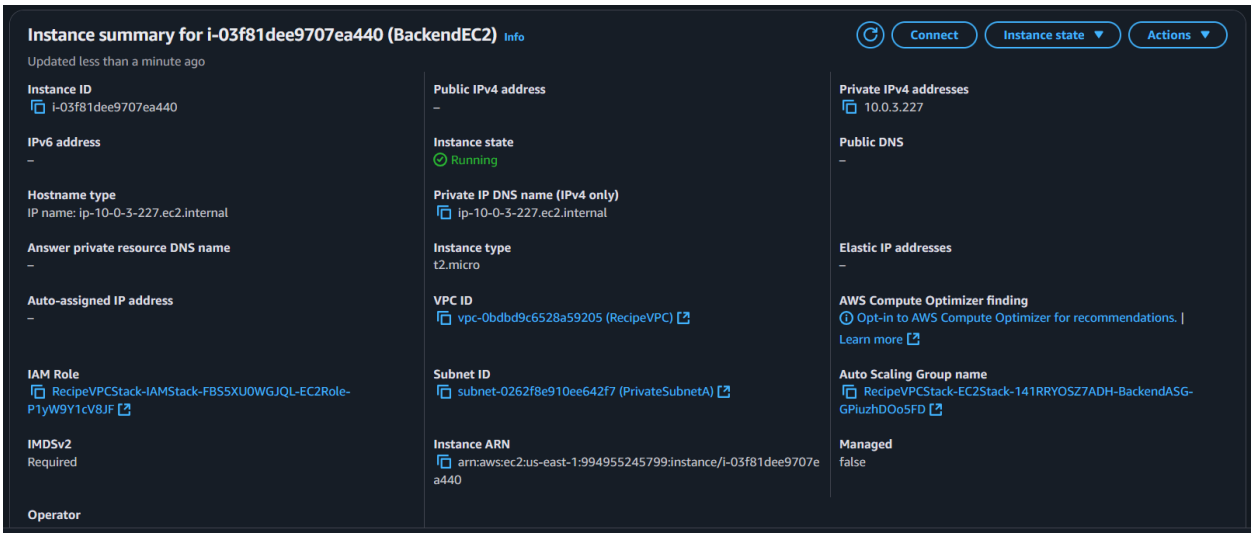

Figure 9 : Backend EC2

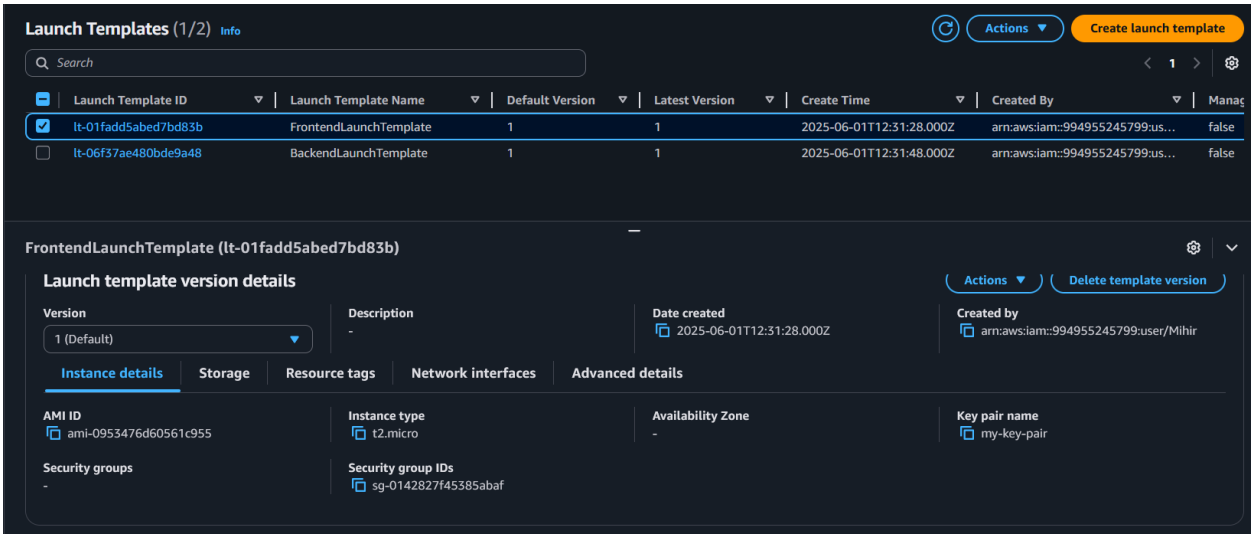

Figure 10 : Launch Templates

Application Load Balancers (ALBs):

I set up two ALBs across two AZs—one frontend and one backend. The frontend ALB listens on HTTP (port 80), routing traffic to a target group with the frontend EC2 on port 80. The backend ALB

listens on HTTP (port 5000), routing to a target group with the backend EC2 on port 5000. I enable access logs, storing them in an S3 bucket for monitoring.

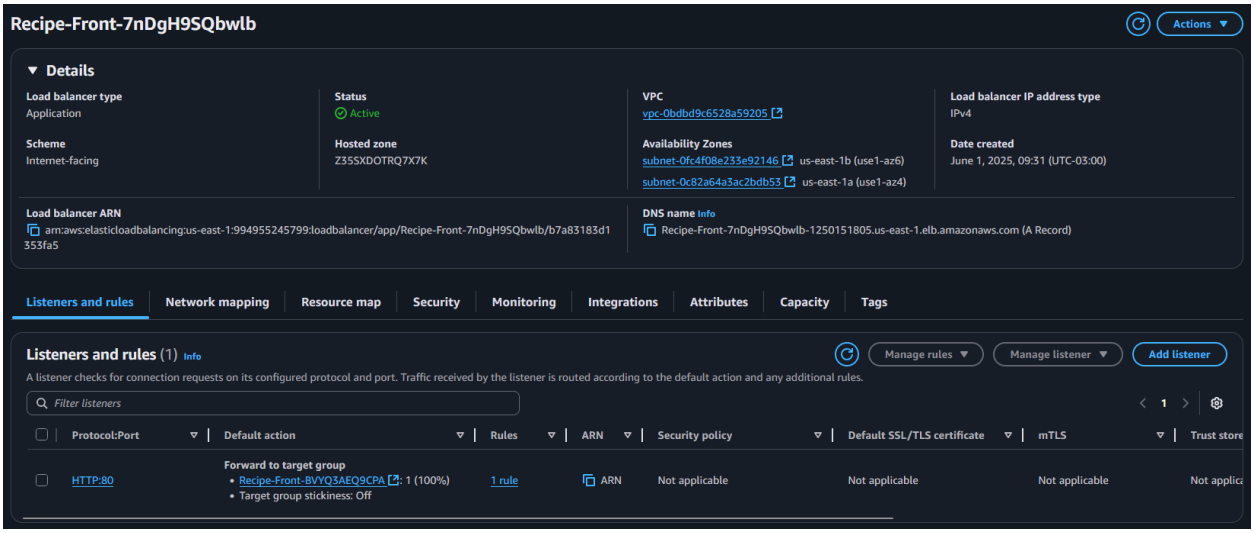

Figure 11 : Frontend Application Load Balancer

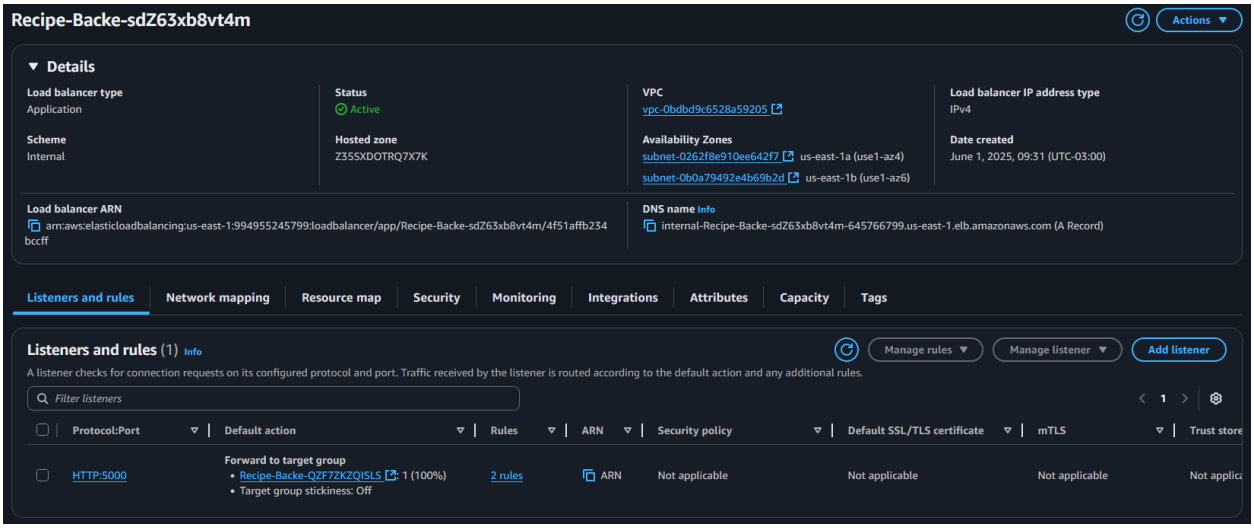

Figure 12 : Backend Application Load Balancer

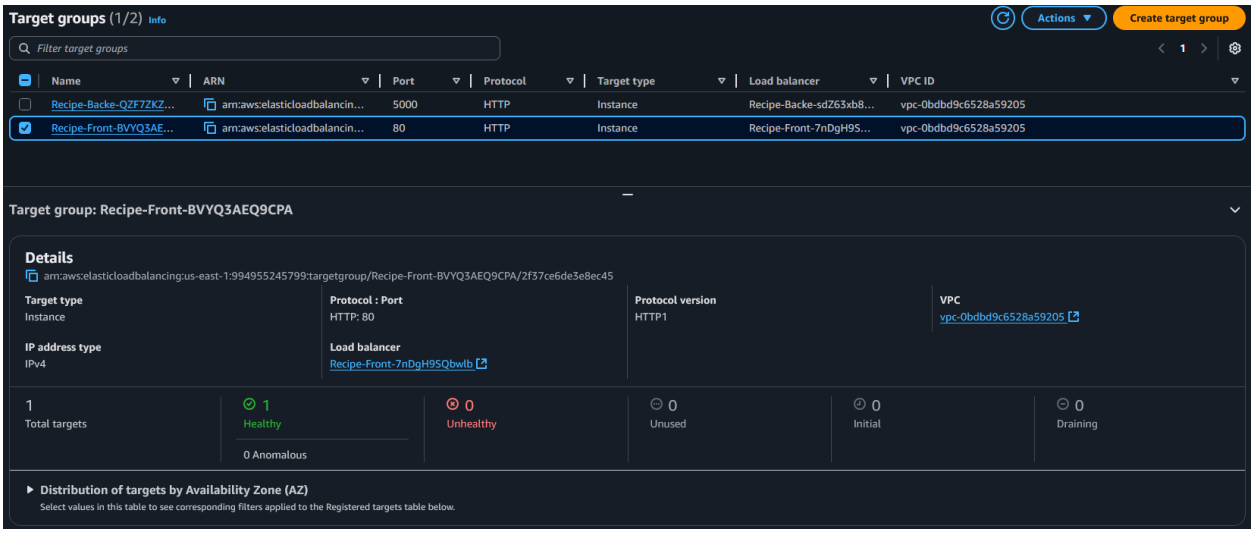

Figure 13 : Target groups

Auto Scaling Group:

I configure two ASGs—one for the frontend and one for the backend—each with a minimum of 1, maximum of 2, and desired capacity of 1 instance. I link each ASG to its respective launch template, ensuring scalability with t2.micro instances. I monitor ASG health via the ALBs.

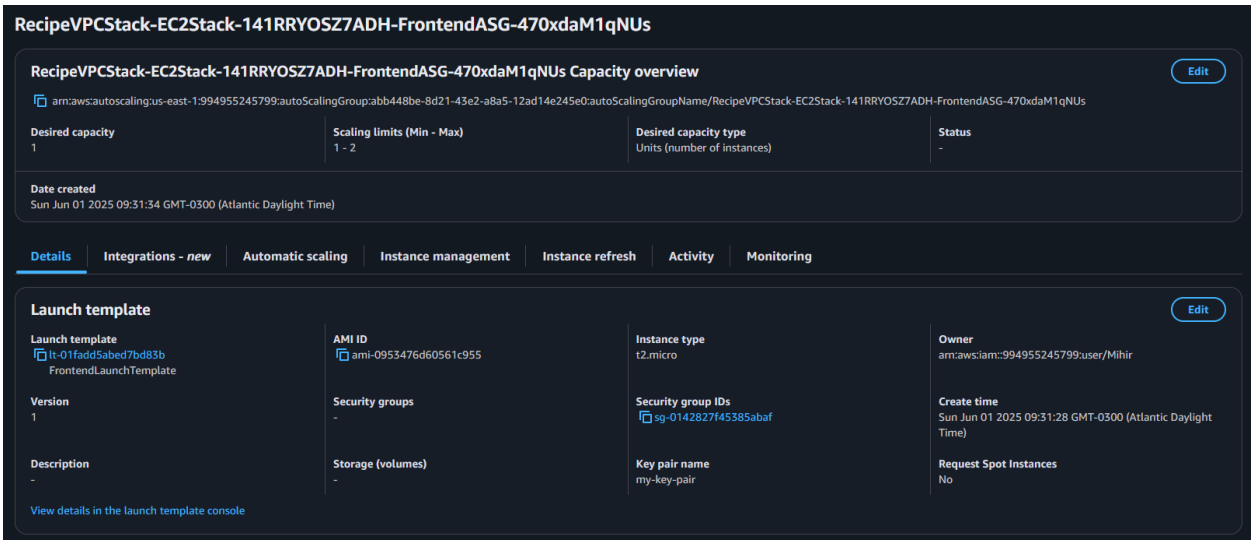

Figure 14 : Frontend Auto Scaling Group

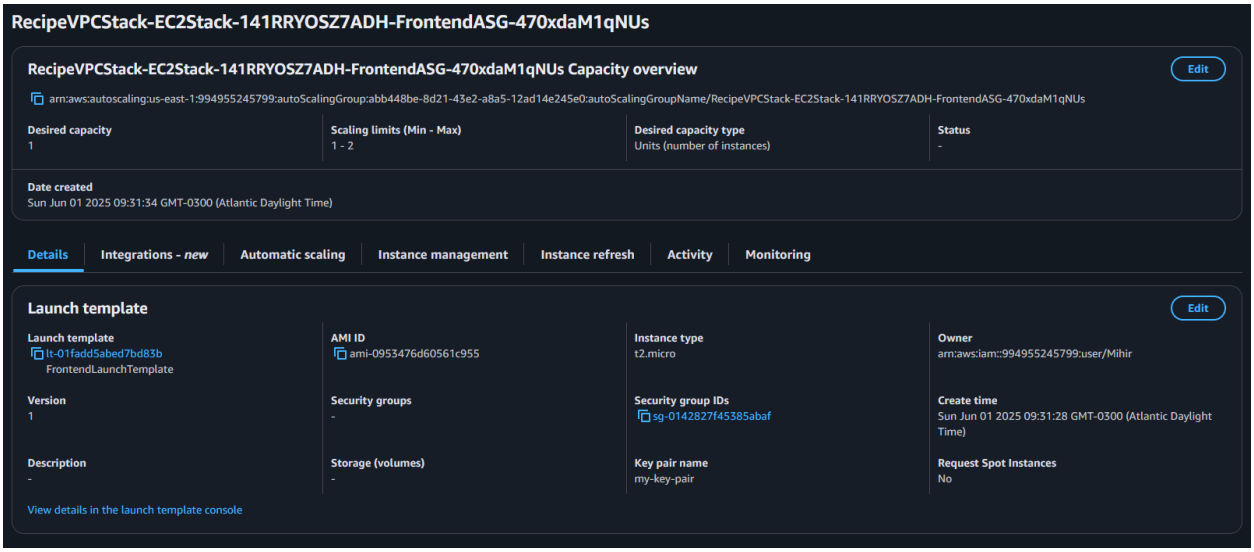

Figure 15 : Backend Auto Scaling Group

OpenSearch:

I establish an OpenSearch domain (t2.small.search, free tier, 10GB EBS, single-AZ) in a private subnet, restricting access to the backend EC2 role (arn:aws:iam::<AID>:role/OpenSearchEC2Role) via an Access Policy. I initially load data from my local machine using the Python SDK on a public-facing setup, then migrate it by creating a snapshot, uploading it to an S3 bucket with SSE-S3 encryption, and restoring it to the private domain using a temporary EC2 instance.

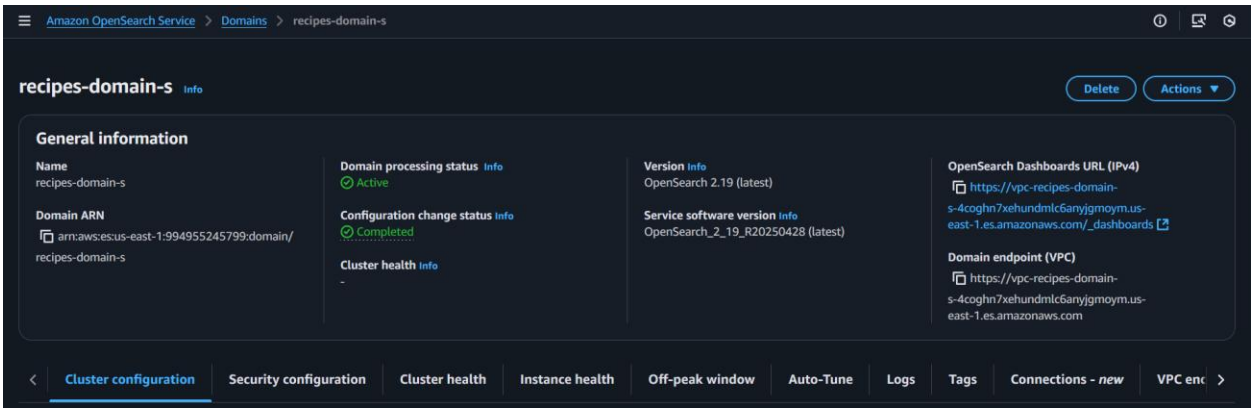

Figure 16 : OpenSearch Domain – 1

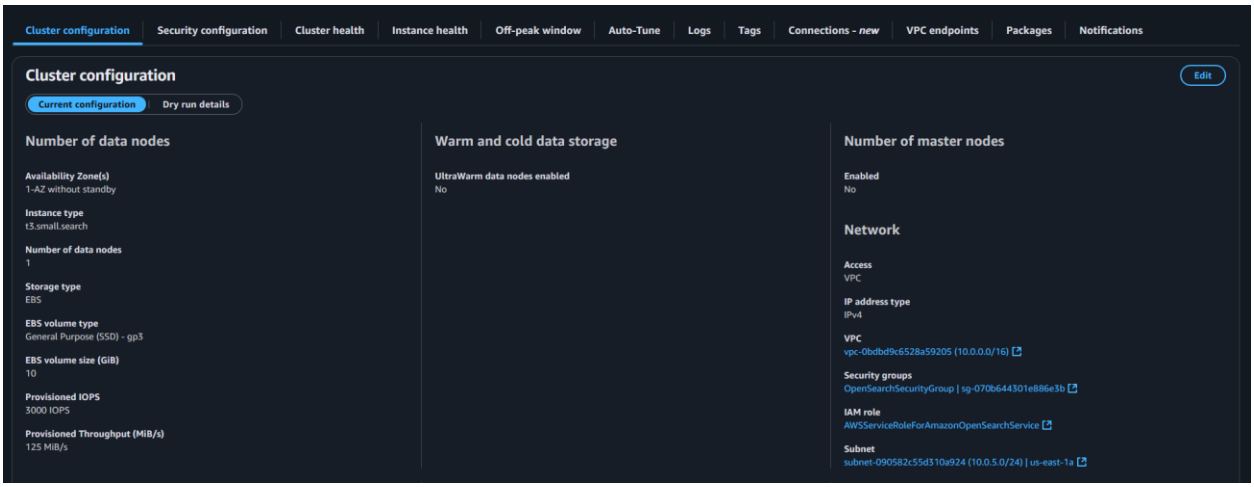

Figure 17 : OpenSearch Domain – 2

SageMaker:

I create two endpoints using the Python SDK: all-MiniLM-L6-v2 on ml.t2.medium for vector conversion and sshleifer/distilbart-cnn-6-6 on ml.m5.xlarge for recipe generation. I configure them as public-facing due to their open-source nature, enable CloudWatch logging, and delete them when idle to optimize costs.

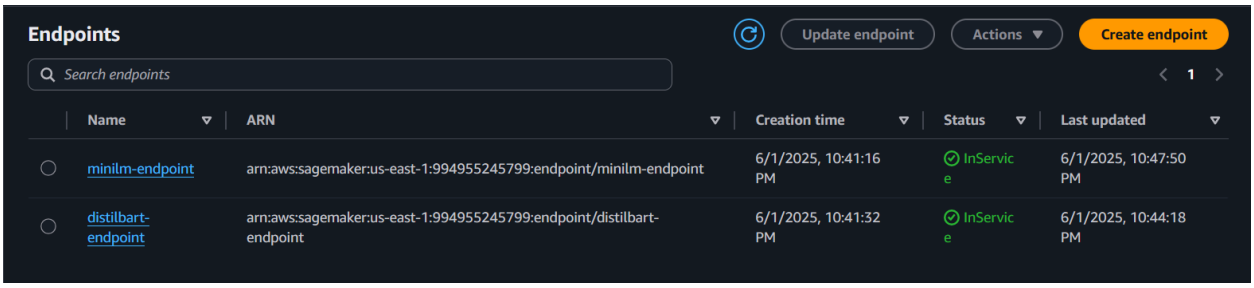

Figure 18 : SageMaker endpoints

---

4. **Implementation Details**

Service Configuration

- **Frontend and Backend Setup**: The frontend ALB routes traffic to frontend EC2 instances (Nginx, t2.micro) in a public subnet, though I could place them in a private subnet for added security, opting for simplicity given the app's basic needs. The backend ALB routes traffic to backend EC2 instances (Flask, t2.micro) in a private subnet. Security groups restrict access: users can only access the frontend ALB, which communicates with frontend EC2, backend ALB, backend EC2, OpenSearch, and SageMaker in a chained manner.

- **SageMaker Configuration**: Two endpoints were configured using the Python SDK: all-MiniLM-L6-v2 [3] (ml.t2.medium) for vector conversion and sshleifer/distilbart-cnn-6-6 [4] (ml.m5.xlarge) for recipe generation. The endpoints are deleted when not in use to save costs and relaunched via SDK during testing.
- **OpenSearch Configuration**: I configured OpenSearch to host the recipe index with 384-dimensional vectors. Initially public, I migrated it to a private subnet by processing data locally, uploading it via Python script, creating a snapshot, uploading it to S3 with SSE-S3 encryption, launching a private domain, and restoring the snapshot using a temporary EC2 instance with S3 and OpenSearch access. I applied an Access Policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": [
          "arn:aws:iam::<ACID>:role/OpenSearchEC2Role",
          "arn:aws:iam::<ACID>:user/Mihir"
        ]
      },
      "Action": "es:*",
      "Resource": "arn:aws:es:us-east-1:<ACID>:domain/recipes-domain-s/*"
    }
  ]
}
```

- **S3**: Used to store OpenSearch snapshots and IaC [6] scripts, encrypted with server-side SSE-S3.

Model Selection, Training, and Integration

- **Model Selection**: I initially selected t5-small for recipe generation due to its lightweight design, but its weak text-generation capabilities produced inconsistent outputs. I switched to sshleifer/distilbart-cnn-6-6 [4] on an ml.m5.xlarge instance, leveraging its strong summarization and generation skills to improve recipe quality. For vector conversion, I chose all-MiniLM-L6-v2 [3] on an ml.t2.medium instance, valuing its ability to efficiently produce embeddings optimized for OpenSearch KNN searches.
- **Techniques**: I implemented Retrieval-Augmented Generation (RAG) [1] by storing recipe vectors in OpenSearch's recipe index, enabling fast retrieval of relevant data based on user input. I applied prompt engineering to craft context-rich prompts, combining retrieved recipe details with user queries to guide the LLM effectively. I utilized in-context learning by feeding these prompts to sshleifer/distilbart-cnn-6-6, allowing it to generate tailored recipes. The recipe index features a 384-dimensional vector field for KNN [5] search, alongside text fields for title, ingredients, directions, and a keyword source field, ensuring comprehensive data handling.

- **Integration**:
    - integrate the SageMaker all-MiniLM-L6-v2 output by querying OpenSearch with KNN to retrieve the top five relevant recipes based on vector similarity.
    - The backend Flask app processes this data, constructs a detailed prompt using retrieved titles, ingredients, and directions, and sends it to the sshleifer/distilbart-cnn-6-6 endpoint for recipe generation.
    - It returns a JSON response with the generated recipe, OpenSearch relevance scores, and source links to original recipes, boosting user trust.
    - I test this end-to-end flow with a local Python script, connecting to live SageMaker endpoints and OpenSearch to confirm real-time performance and data consistency.
- **Demonstration**: I deployed the app at http://recipe-front-7ndgh9sqbwlb-1250151805.us-east-1.elb.amazonaws.com/ . I'll provide an updated URL in the one-on-one meeting if it changes.

Monitoring and Logging Solutions

CloudWatch logs are implemented to monitor SageMaker endpoints, capturing metrics like invocation errors and latency. However, due to time constraints, no alerts or dashboards were set up. Future iterations could include a CloudWatch alarm for SageMaker endpoint latency (> 40 sec) and a dashboard with widgets for SageMaker invocation errors and OpenSearch query performance.

Testing Methodology and Results

- **SageMaker-OpenSearch Interaction**: I used a local Python script to test the RAG flow, simulating user input, converting it to vectors with SageMaker, querying OpenSearch for relevant data, and generating a recipe via the LLM. The script validated the end-to-end workflow by connecting to live AWS endpoints, ensuring seamless integration.
- **Performance Testing**: I conducted initial tests with sshleifer/distilbart-cnn-6-6 on ml.t2.medium, recording a 2-minute average response time due to resource constraints. I optimized this by upgrading to ml.m5.xlarge, reducing the average to 30 seconds for faster user feedback.
- **Results**: I configured OpenSearch KNN queries to return relevance scores (e.g., _score from client.search()), which I display to users alongside related recipes and source links for transparency. The generated recipes lack full accuracy due to the smaller LLM, but I enhance usability by including reliable references, and I resolved initial errors by optimizing instance sizes.

---

5. **Security Measures**

Implemented Security Measures

- **VPC and Subnets**: I place all resources except the frontend ALB and SageMaker endpoints in private subnets within a VPC. I migrated OpenSearch to a private subnet and restrict its access with an Access Policy to the backend EC2 role (arn:aws:iam::<ACID>:role/OpenSearchEC2Role).
- **Security Groups**: I configure security groups to allow user access only to the frontend ALB, enabling chained access (Frontend ALB → Frontend EC2 → Backend ALB → Backend EC2 → OpenSearch → SageMaker).
- **Encryption**: I ensure S3 buckets storing OpenSearch snapshots and IaC scripts use server-side SSE-S3 encryption.
- **IAM Policy**: I restrict OpenSearch access to the backend EC2 role, securing data retrieval.

Unimplemented Security Measures

The SageMaker endpoints remain public-facing due to their open-source models and lack of confidential data. In a non-restricted environment, they could be placed in a private subnet with VPC endpoints for added security.

Justification

The implemented measures align with best practices by isolating critical resources in private subnets, restricting access with IAM policies and access policies, and ensuring encrypted data storage. The public SageMaker endpoints pose minimal risk, as they host open-source models without sensitive data.

---

6. **Cost Analysis and Optimization**

Cost Breakdown (30 days):

- **EC2 (2 x t2.micro):**
    - **Free Tier:** 750 hours/month (covers 1 instance fully).
    - **Cost:** 1,440 hours (2 × 24 × 30) - 750 = 690 hours × $0.0104/hour = $7.18.
    - **Total**: $7.18.
- **2 Application Load Balancers (ALBs):**

    - **Cost:** 2 × $0.0225/hour × 24 × 30 + 2 × $0.008/LCU-hour × 30 = $32.40 + $0.48 = $32.88.
    - **Total**: $32.88.
- **Amazon OpenSearch (t2.small.search, single-AZ):**
    - **Cost**: $0.036/hour × 24 × 30 = $25.92.
    - **Total**: $25.92 (free tier utilized during initial setup, actual cost reflects full usage).
- **Amazon SageMaker (ml.t2.medium and ml.m5.xlarge):**
    - **Free Tier**: 250 hours/2 months (covers 250 hours).
    - **Cost**: all-MiniLM-L6-v2 (ml.t2.medium) - 720 hours (24 × 30) - 250 = 470 hours × $0.046/hour = $21.62.
    - **Cost**: sshleifer/distilbart-cnn-6-6 (ml.m5.xlarge) - 720 hours × $0.23/hour = $165.60 (active only during testing, assumed 100 hours) = $23.00.
    - **Total**: $21.62 + $23.00 = $44.62.
- **Amazon S3 (1.51 GB, 1,000 requests):**
    - **Free Tier**: 5 GB, 20,000 GET requests/month.
    - **Cost**: $0 (within Free Tier).
    - **Total**: $0.
- **NAT Gateway (us-east-1a):**
    - **Cost**: $0.045/hour × 24 × 30 × 0.5 (50% utilization) = $16.20.
    - **Total**: $16.20.
- **Amazon CloudWatch (10 metrics, 1 GB logs):**
    - **Free Tier**: 10 metrics, 5 GB logs/month.
    - **Cost**: $0 (within Free Tier).
    - **Total**: $0.
- **Total Estimated Cost (30 days):**
    - $7.18 (EC2) + $32.88 (ALBs) + $25.92 (OpenSearch) + $44.62 (SageMaker) + $0 (S3) + $16.20 (NAT Gateway) + $0 (CloudWatch) = **$126.80**.

Optimization Strategies

- **SageMaker Endpoints**: Used the Python SDK to delete endpoints when not in use, relaunching them during testing to minimize costs.

- **Instance Selection**: Leveraged free-tier instances where possible: t2.micro for EC2, t2.small.search for OpenSearch, and ml.t2.medium for vector generation. Only the recipe generation endpoint used a larger instance (ml.m5.xlarge) to improve performance.
- **IaC and SDK**: Shifted to Infrastructure as Code (IaC) and SDKs early in development to automate resource creation and deletion, reducing costs by ensuring resources were only active when needed.

Free Trial Usage

Using a personal AWS account with a free trial, no significant challenges were faced with resource usage. The combination of IaC, SDKs, and free-tier instances ensured cost efficiency.

---

7. **Lessons Learned and Future Improvements**

Lessons Learned

- **Cost Management**: Initially creating resources manually led to potential cost overruns. Shifting to IaC and SDKs within the first few days saved costs by automating resource lifecycle management. I will prioritize IaC from the start in future cloud projects.
- **Monitoring Importance**: The lack of robust monitoring (e.g., CloudWatch dashboards, alerts) made troubleshooting difficult when SageMaker instances crashed on smaller instances. I recognize that implementing monitoring earlier would improve debugging.

Future Improvements

- **Larger Model**: Use a more powerful LLM for recipe generation to improve accuracy and realism of outputs.
- **User Features**: Add a backend to store user history, favorite ingredients, and recipes, potentially reintroducing Cognito for authentication.
- **Enhanced Monitoring**: Implement CloudWatch dashboards and alerts to monitor SageMaker latency and OpenSearch query traffic, improving system reliability.

---

8. **Conclusion**

This project successfully demonstrates the development and deployment of an AI/GenAI application on AWS, focusing on RAG, prompt engineering, and in-context learning. The architecture adheres to best practices, with secure resource isolation, cost optimization, and monitoring via CloudWatch logs. The application, accessible at http://recipe-front-7ndgh9sqbwlb-1250151805.us-east-1.elb.amazonaws.com/ , provides a functional recipe suggestion system, validated by a 30-second response time and relevance scores. Despite using a smaller LLM, the inclusion of related recipes and links enhances reliability, meeting the course objectives of cloud-based AI solution design.

References

[1] Amazon Web Services, "What is Retrieval-Augmented Generation (RAG)? – AWS," *aws.amazon.com*. Accessed: Jun. 1, 2025. [Online]. Available: https://aws.amazon.com/what-is/retrieval-augmented-generation/

[2] Cloudflare, "What is a vector database? | Cloudflare," *Cloudflare*. Accessed: Jun. 1, 2025. [Online]. Available: https://www.cloudflare.com/en-ca/learning/ai/what-is-vector-database/

[3] sentence-transformers, "sentence-transformers/all-MiniLM-L6-v2," *Hugging Face*. Accessed: Jun. 1, 2025. [Online]. Available: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

[4] sshleifer, "sshleifer/distilbart-cnn-6-6," *Hugging Face*. Accessed: Jun. 1, 2025. [Online]. Available: https://huggingface.co/sshleifer/distilbart-cnn-6-6

[5] IBM, "What is KNN?," *IBM*. Accessed: Jun. 1, 2025. [Online]. Available: https://www.ibm.com/think/topics/knn

[6] Amazon Web Services, "What is Infrastructure as Code (IaC)? – AWS," *aws.amazon.com*. Accessed: Jun. 1, 2025. [Online]. Available: https://aws.amazon.com/what-is/iac/

[7] M. Patel, "Data_sequence_diagram.drawio." Accessed: Jun. 1, 2025. [Online]. Available: https://drive.google.com/file/d/185eFIRzNVwFpWSsifb2F_C7AL13p4het/view?usp=sharing

[8] M. Patel, "Recipe_Architecture.drawio." Accessed: Jun. 1, 2025. [Online]. Available: https://drive.google.com/file/d/1CQTsUrgjjT-Uqt2ScevvSjGaRkbJtiDp/view?usp=sharing