Smart-Kitchen-Helper Project Documentation

# 1. Team Members

* **Amit Christian: 100937637** - Project Lead, Frontend Developer
* **Sagar Parmar: 100942558** - Backend Developer
* **Shukan Thakkar: 100944139** - Database Specialist
* **Jinal Bambhroliya: 100964562** - API Integration and Security Specialist
* **Arundarasi Rajendran: 100946579** - Cloud Infrastructure and Networking Engineer

# 2. Introduction

## 2.1 Project Overview

The “Smart-Kitchen-Helper” project is a cloud-based web application designed to streamline the kitchen management process. It allows users to track and manage their household ingredients, search for recipes that match their available supplies, and access multimedia content such as recipe videos to assist in meal preparation. The project leverages Amazon Web Services (AWS) to ensure the system is scalable, highly available, and secure, offering users a seamless and reliable experience.

This application was conceived in response to the growing need for efficient kitchen management solutions in modern households. With a focus on reducing food waste, enhancing meal planning, and providing easy access to cooking resources, the “Smart-Kitchen-Helper” aims to become an essential tool for home cooks.

## 2.2 Objectives

The primary objectives of the “Smart-Kitchen-Helper” project include: - **Simplified Ingredient Management:** Enable users to store, manage, and track their household ingredients with ease, including monitoring quantities, expiration dates, and usage history. - **Intelligent Recipe Discovery:** Allow users to search for recipes based on the ingredients they have on hand, reducing food waste and encouraging creative cooking. - **Enhanced Cooking Experience:** Integrate multimedia content, such as video tutorials, to guide users through the cooking process. - **Scalability and Reliability:** Ensure that the application can scale to accommodate increasing user demands while maintaining high availability and performance. - **Robust Security:** Implement strong security measures to protect user data and ensure compliance with industry best practices.

# 3. Background

## 3.1 Scenario

The idea for the “Smart-Kitchen-Helper” project emerged from the common challenges faced by households in managing kitchen resources. In today’s fast-paced world, many people struggle with keeping track of their ingredients, planning meals, and finding recipes that match what they have available. These challenges often lead to food waste, missed opportunities to create nutritious meals, and frustration in the kitchen.

As part of a broader initiative to promote smart home solutions, our team was tasked with developing a cloud-based application that would address these challenges. The goal was to create an intuitive platform that would help users make the most of their kitchen resources by providing intelligent recipe suggestions, automated ingredient tracking, and multimedia support.

## 3.2 Technical Challenges

The development of the “Smart-Kitchen-Helper” application involved overcoming several technical challenges:

* **Scalability:** The system needed to handle varying traffic loads, especially during peak usage times, without compromising performance. AWS Auto Scaling and load balancing were implemented to dynamically adjust resources based on demand.
* **Data Management:** Efficiently storing and retrieving both structured data (e.g., user profiles, ingredient inventories) and unstructured data (e.g., images, videos) was essential. The team chose RDS PostgreSQL for structured data and MongoDB Atlas for unstructured data to meet these needs.
* **Security:** Protecting user data was a top priority. The application was designed with multiple layers of security, including encryption, secure access controls, and continuous monitoring to detect and respond to potential threats.
* **Integration:** Integrating with external APIs, such as YouTube, posed challenges in terms of managing API rate limits, ensuring data consistency, and providing a seamless user experience. The team used AWS Lambda to manage these integrations efficiently.

# 4. Rationale

## 4.1 Design Methodologies

The design of the “Smart-Kitchen-Helper” application was guided by several key methodologies to ensure that it meets the project objectives and technical requirements:

* **Microservices Architecture:** The application was developed using a microservices architecture, where each service is responsible for a specific functionality (e.g., user management, ingredient tracking, recipe search). This approach enhances the scalability and maintainability of the application. Each microservice can be developed, deployed, and scaled independently, allowing the team to make updates and improvements without affecting the entire system.
* **Cloud-Native Design:** The application was built with a cloud-native approach, leveraging AWS services such as EC2, RDS, and Lambda. This ensures that the application can take full advantage of the cloud’s scalability, resilience, and security features. By using cloud-native services, the team was able to focus on developing the application without worrying about the underlying infrastructure.
* **Data Security:** Data security was a critical consideration in the design of the application. The team implemented encryption for data at rest and in transit, using AWS Key Management Service (KMS) for managing encryption keys. Additionally, secure access controls were established using AWS Identity and Access Management (IAM) to ensure that only authorized users and services could access sensitive data.
* **Serverless Computing:** AWS Lambda was used to implement the integration with the YouTube API for fetching recipe videos. By using serverless computing, the team was able to reduce operational overhead, as Lambda automatically scales to handle incoming requests and only incurs costs when the function is executed. This approach also allowed the team to quickly develop and deploy the video retrieval functionality without worrying about server management.

## 4.2 Reasons Behind Design Choices

* **AWS as the Cloud Provider:** AWS was selected as the cloud provider due to its comprehensive suite of services, global reach, and robust security features. AWS provides the flexibility and scalability needed to support the “Smart-Kitchen-Helper” application, along with a wide range of tools and services that simplify development and deployment.
* **Multi-AZ Deployment:** The decision to deploy the EC2 instances and RDS PostgreSQL database across multiple availability zones (AZs) was made to ensure high availability and resilience. In the event of an AZ failure, the application can continue to operate without interruption, as traffic is automatically redirected to healthy instances in other AZs.
* **Use of MongoDB Atlas:** MongoDB Atlas was chosen for its ability to handle unstructured data, such as images and JSON files, which are critical components of the application. MongoDB’s flexible schema design allows the team to store and retrieve data in a way that is optimized for the application’s needs. Additionally, MongoDB Atlas provides automated backups, scaling, and monitoring, reducing the operational burden on the team.
* **NAT Gateway:** The NAT Gateway was implemented to provide secure internet access for EC2 instances in private subnets. This design choice minimizes the attack surface by keeping critical components isolated from the public internet, while still allowing necessary outbound traffic, such as API requests and software updates.

# 5. Results/Body/Artefact Produced

## 5.1 Produced Architecture

The architecture of the “Smart-Kitchen-Helper” project was designed to meet the project’s objectives of scalability, high availability, and security. The architecture consists of the following key components:

### 5.1.1 Compute Layer

* **EC2 Instances:** Two EC2 instances are used to host the backend (Node.js) and frontend (Python Flask) applications. These instances are deployed across multiple availability zones to ensure high availability. The Auto Scaling group automatically adjusts the number of instances based on traffic patterns and CPU utilization, ensuring that the application can handle varying workloads.
* **Auto Scaling:** The Auto Scaling group is configured to add or remove EC2 instances based on predefined metrics, such as CPU utilization and request counts. This ensures that the application can scale dynamically to handle traffic spikes, while also reducing costs by terminating instances when demand decreases.
* **Instance Types:** The EC2 instances are of the t3.medium type, which provides a balance of compute, memory, and network resources. This instance type was chosen for its cost-effectiveness and ability to handle the application’s workload.

### 5.1.2 Storage Layer

* **RDS PostgreSQL:**
  + **Database Configuration:** The RDS PostgreSQL instance is configured with Multi-AZ deployment to ensure high availability and automatic failover. The database stores structured data, such as user profiles, ingredient inventories, recipes, and transaction histories.
  + **Performance Tuning:** The database is optimized for performance with features like automated backups, read replicas for scaling read-heavy workloads, and encrypted storage using AWS KMS (Key Management Service). The database schema is designed to be highly normalized, reducing data redundancy and improving query efficiency.
* **MongoDB Atlas:**
  + **NoSQL Database:** MongoDB Atlas is used to store unstructured data, including images of ingredients, video links, and JSON files containing user preferences and ingredient usage patterns. The database is sharded to distribute data across multiple nodes, ensuring high availability and scalability.
  + **Cluster Configuration:** The MongoDB cluster is set up with replication across multiple regions, providing disaster recovery capabilities. MongoDB Atlas also provides automated backups, monitoring, and scaling, reducing the operational burden on the team.

### 5.1.3 Network Layer

* **VPC (Virtual Private Cloud):**
  + **Custom VPC:** A custom VPC is created to isolate the application environment, with subnets divided into public and private categories. Public subnets are used for the Load Balancer and NAT Gateway, while private subnets house the EC2 instances and RDS database. This separation ensures that the internal components are shielded from direct exposure to the internet, enhancing security.
  + **CIDR Block:** The VPC is assigned a CIDR block of 10.0.0.0/16, with subnets further divided into smaller blocks to accommodate different components. This CIDR block allows for a large number of IP addresses, ensuring that the network can scale as needed.
  + **Subnets:** Two public subnets (one in each AZ) and two private subnets are created, with routing tables configured to manage traffic flow efficiently. The public subnets are used for internet-facing resources, while the private subnets are used for resources that do not require direct internet access.
* **NAT Gateway:**
  + **Internet Access:** A NAT Gateway is deployed in a public subnet to provide internet access to the EC2 instances in private subnets. This setup ensures that the instances can access external services, such as the YouTube API, without being exposed to direct internet access.
  + **High Availability:** The NAT Gateway is designed to be highly available, with automatic failover in case of failures. It is associated with an Elastic IP address, ensuring consistent access to the internet.
* **Load Balancer:**
  + **Application Load Balancer (ALB):** An ALB is used to distribute incoming traffic across the EC2 instances. The ALB performs health checks to ensure that traffic is routed only to healthy instances, improving the overall reliability of the application.
  + **SSL Termination:** The ALB is configured with SSL termination to offload the SSL decryption from the backend instances, reducing their processing load. This also ensures that all communication between users and the application is encrypted.

### 5.1.4 Security

* **Security Groups:**
  + **Inbound and Outbound Rules:** Security groups are configured to control traffic to and from the EC2 instances, RDS database, and MongoDB Atlas. Inbound rules restrict access to specific IP ranges and ports, while outbound rules allow only necessary traffic to leave the network.
  + **Instance Isolation:** Each component (EC2, RDS, MongoDB) is assigned its own security group to isolate traffic and minimize the risk of unauthorized access.
* **IAM Roles and Policies:**
  + **Granular Permissions:** IAM roles and policies are defined to grant the least privilege necessary for each service to function. For example, the Lambda function is granted access only to the YouTube API and MongoDB Atlas, while the EC2 instances have permissions to access the RDS database and S3 buckets.
  + **MFA and Role Segregation:** Multi-Factor Authentication (MFA) is enforced for administrative access, and roles are segregated based on function (e.g., application, database, monitoring).
* **Encryption:**
  + **Data Encryption:** Data at rest in RDS PostgreSQL and MongoDB Atlas is encrypted using AWS KMS. Data in transit is secured with SSL/TLS, ensuring that all communication between services is encrypted and secure.
  + **Backup Encryption:** Automated backups and snapshots of RDS PostgreSQL and MongoDB Atlas are encrypted to prevent unauthorized access to backup data.

### 5.1.5 Integration Layer

* **AWS Lambda:**
  + **Serverless Function:** An AWS Lambda function is used to fetch recipe videos from the YouTube API. The function is triggered by user requests to ensure that the most relevant and up-to-date video content is retrieved.
  + **API Integration:** The Lambda function calls the YouTube Data API using the user’s search query (e.g., “Pasta Carbonara Recipe Video”) and returns a video link to the front-end application.
* **YouTube API:**
  + **Dynamic Video Fetching:** The YouTube API is integrated into the application to provide users with instructional videos for their chosen recipes. The API is accessed via the Lambda function, ensuring that users receive video content tailored to their search.
  + **API Rate Limiting:** The application includes logic to handle API rate limits, ensuring that API requests are managed efficiently without exceeding YouTube’s quota limits.

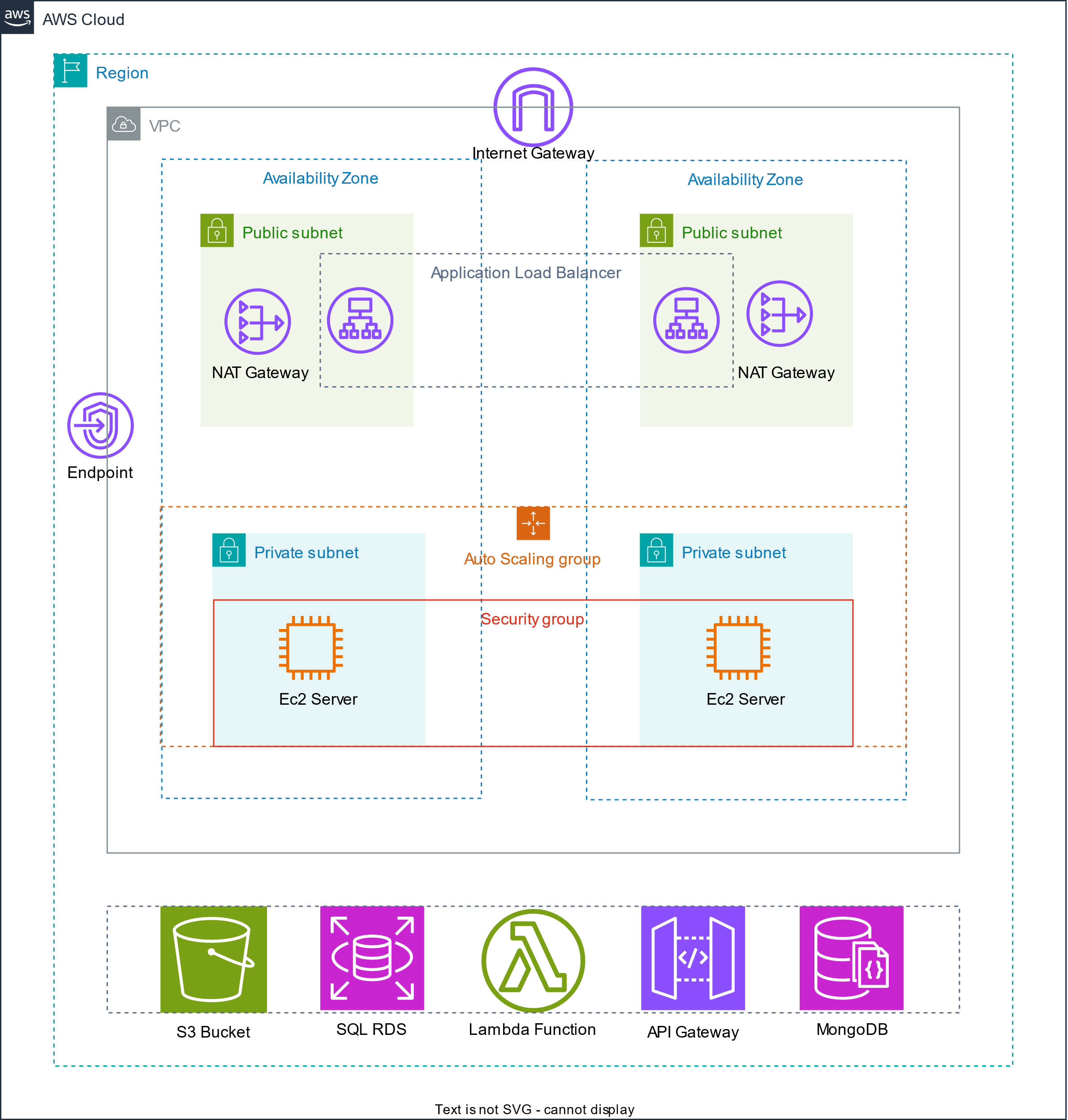
### 5.1.6 Web End-Point

* **Web Application:**
  + **Frontend Development:** The web application is developed using modern web technologies, including HTML5, CSS3, and JavaScript (React.js for the user interface). The frontend is designed to be responsive and user-friendly, with features like ingredient management, recipe search, and video playback.
  + **Backend APIs:** The frontend communicates with the backend services via RESTful APIs. These APIs are developed using Node.js and Flask, providing endpoints for user management, ingredient tracking, recipe discovery, and video retrieval.
  + **User Authentication:** The application uses JSON Web Tokens (JWT) for secure user authentication. The JWT tokens are issued by the backend and stored in the user’s browser, enabling secure session management.

## 5.2 Diagrams and Descriptions

The following diagrams provide a detailed view of the overall design, network topology, data flow, and integration points:

**Figure 1: AWS Architecture Placeholder**



**Figure** **:** AWS Architecture Diagram

This placeholder represents the architecture of the Smart-Kitchen-Helper project, which includes key AWS components such as EC2 instances, RDS PostgreSQL, MongoDB Atlas, and Load Balancer. The diagram highlights how these components interact within the VPC to provide a secure and scalable environment for the application.

**Figure 2: Index.html**

A screenshot of a website

Description automatically generated

**Figure** **:** Index.html

This placeholder represents the design of the index.html page for the Smart-Kitchen-Helper web application. The page includes features like user login, ingredient management, and recipe search. The design is responsive, ensuring a seamless user experience across different devices.

## 5.3 Implementation Details

### 5.3.1 Setting Up the VPC

1. **Create a Custom VPC:**
   * Use the AWS Management Console or AWS CLI to create a custom VPC with a CIDR block of 10.0.0.0/16.
   * Create two public subnets (e.g., 10.0.1.0/24 and 10.0.2.0/24) and two private subnets (e.g., 10.0.3.0/24 and 10.0.4.0/24), each in different availability zones.
2. **Configure Route Tables:**
   * Create a route table for the public subnets and associate it with the Internet Gateway (IGW) for internet access.
   * Create a route table for the private subnets and associate it with the NAT Gateway to allow outbound internet access without direct exposure to the internet.

### 5.3.2 Deploying EC2 Instances

1. **Launch EC2 Instances:**
   * Use the AWS Management Console to launch two t3.medium EC2 instances in the private subnets.
   * Configure the instances to run the Node.js and Python Flask applications, and set up Auto Scaling to manage instance scaling automatically.
2. **Set Up Security Groups:**
   * Create security groups for the EC2 instances, allowing inbound traffic on port 80 (HTTP) and port 443 (HTTPS), and restricting outbound traffic as needed.
   * Ensure that the security group allows traffic from the ALB and other necessary services.

### 5.3.3 Configuring RDS PostgreSQL

1. **Create an RDS Instance:**
   * Launch an RDS PostgreSQL instance with Multi-AZ deployment.
   * Configure automated backups, enable encryption, and set up read replicas if needed.
2. **Database Schema Setup:**
   * Use tools like pgAdmin or the psql command line to create the necessary tables for users, households, ingredients, and recipes.
   * Optimize the database schema for performance, including indexing frequently queried columns.

### 5.3.4 Setting Up MongoDB Atlas

1. **Create a MongoDB Cluster:**
   * Set up a MongoDB Atlas cluster with sharding to handle large datasets and ensure high availability.
   * Configure replication across multiple regions for disaster recovery.
2. **Integrate MongoDB with the Application:**
   * Use Mongoose or another MongoDB client to connect the application to MongoDB Atlas.
   * Store images, videos, and JSON files in MongoDB, ensuring data is properly indexed for quick retrieval.

### 5.3.5 Implementing NAT Gateway

1. **Deploy NAT Gateway:**
   * Create a NAT Gateway in one of the public subnets and associate it with the Elastic IP.
   * Update the private subnet’s route table to route internet-bound traffic through the NAT Gateway.
2. **Test Internet Access:**
   * SSH into an EC2 instance in the private subnet and verify that it can access the internet via the NAT Gateway.

### 5.3.6 Deploying Lambda Function for YouTube API

1. **Write the Lambda Function:**
   * Use AWS Lambda to create a function that calls the YouTube Data API based on user queries.
   * Implement error handling, logging, and API rate limiting within the function.
2. **Trigger Lambda from the Application:**
   * Configure the application to invoke the Lambda function when a user searches for a recipe video.
   * Ensure the function’s response is properly formatted and integrated into the frontend.

### 5.3.7 Setting Up Load Balancer

1. \*\*Create an Application

Load Balancer (ALB):\*\* - Deploy an ALB in the public subnets to distribute incoming traffic to the EC2 instances. - Configure SSL certificates for HTTPS traffic and set up health checks to monitor instance health.

1. **Integrate ALB with EC2 Instances:**
   * Register the EC2 instances with the ALB and verify that the traffic is routed correctly.
   * Test the load balancing by simulating traffic and observing how it is distributed across instances.

### 5.3.8 Deploying the Web Application

1. **Develop the Frontend:**
   * Build the frontend using React.js, ensuring that the user interface is responsive and intuitive.
   * Implement features like ingredient management, recipe search, and video playback.
2. **Deploy the Frontend:**
   * Host the frontend on the EC2 instances and configure the backend APIs to handle user requests.
   * Connect the frontend to the backend services and Lambda functions to provide a seamless user experience.
3. **Set Up Continuous Deployment:**
   * Use CI/CD tools like AWS CodePipeline or Jenkins to automate the deployment process, ensuring that updates are deployed quickly and reliably.

## 5.4 Current Progress

* **Frontend Implementation:** The development of the frontend is currently in progress. The main pages, including the homepage, login, and ingredient management pages, have been completed. However, some route implementations for specific paths (e.g., recipe search results and video playback) are still pending.
* **Backend API Development:** The backend APIs for user management, ingredient tracking, and recipe search are mostly implemented. Integration with the Lambda function for fetching recipe videos is in progress and is expected to be completed soon.
* **Database Setup:** The RDS PostgreSQL and MongoDB Atlas databases have been configured and are operational. Data migration scripts are being developed to populate the databases with initial data, such as sample recipes and ingredients.
* **Security Configuration:** Security groups, IAM roles, and encryption settings have been configured. Additional security measures, such as AWS WAF (Web Application Firewall), are being considered to protect the application from common web attacks.
* **Testing and Validation:** Preliminary testing has been conducted on the backend APIs and database interactions. Load testing and security audits are planned for the upcoming development sprints.

# 6. References

All references are formatted according to APA 7th edition:

* Amazon Web Services. (2023). *Amazon RDS Multi-AZ Deployments*. Retrieved from <https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.MultiAZ.html>
* MongoDB. (2023). *Sharding in MongoDB Atlas*. Retrieved from <https://www.mongodb.com/cloud/atlas>
* Google Developers. (2023). *YouTube Data API v3 Documentation*. Retrieved from <https://developers.google.com/youtube/v3>
* Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O’Reilly Media.
* AWS Well-Architected Framework. (2023). *Best Practices for Designing Cloud Applications*. Retrieved from <https://aws.amazon.com/architecture/well-architected/>