Ingredient Lens

CS492 Section A

Spring 2024

Jack Berkowitz

Dmitry Bezborodov

Andrew Catapano

John Costa

Computer Science Program

Monmouth University

Table of Contents

1. Introduction

Ingredient Lens is an artificial intelligence (AI) based application that allows the user to upload a picture of a food dish. Then the AI recognizes the dish in the photo and outputs a recipe on how to make it. This application was implemented using a combination of AWS Services, such as Amplify, S3, Rekognition, and API Gateways, React JS, the OpenAI API, Python code, and Next UI.

1.1 Background

AI has seen a surge in the computer science world, reshaping the way we use technology and creating a more complex future. AI is an extremely powerful tool that can improve all types of industries by empowering machines to perform tasks that were never thought possible. This project's main goal is to identify the capabilities of AI and use those ideas to implement a relatively simple yet useful application.

Ingredient Lens leverages AI's limitless capabilities by taking advantage of AI technologies including computer vision and natural language processing to provide accurate image recognition of dishes and AI-generated recipes to its users. Its purpose is to make it easier for users to cook food that they have never made before. It also reduces the confusion of cooking by providing easy recipes for users to follow. When it comes to achieving this goal, Ingredient Lens has many specific use cases. The first use case can be when the user is sitting at a restaurant and wants to get the recipe of the dish that he or she was served. Another use case can be if the user saw an interesting photo of the dish on social media and would like to learn more about it and how to make it. In these use cases, Ingredient Lens can be used as an effective tool to satisfy the user's curiosity and provide useful cooking information.

1.2 Potential Impact

Ingredient Lens has the potential to provide many benefits to its users. These benefits can improve their lives by teaching them new skills and knowledge that can improve their self-esteem.

The first benefit is improved cooking skills. Since the user can upload an image of any dish and receive a step-by-step recipe on how to make it, they can improve their cooking skills by trying to cook dishes they have never cooked before. Depending on the dish, they could also be introduced to new ingredients or cooking techniques that they have never used.

The second benefit is better cultural knowledge. Ingredient Lens can expose users to the food of many cultures around the world by giving them guides on how to cook foreign dishes. Even if a user doesn't recognize the dish they want to make, the application will recognize it for them, satisfying their curiosity and allowing them to broaden their culinary horizons.

The third benefit is time efficiency. Ingredient Lens can improve time efficiency by allowing the user to upload a photo of a dish to receive a good recipe rather than having them scour the web for one. This will allow them to jump right into gathering ingredients and cooking rather than exhausting themselves searching for the perfect recipe online.

The fourth and final benefit is saving money. Once the user knows how to cook the dishes they could find in restaurants through Ingredient Lens, they no longer have to spend the extra money to go out to a restaurant to get their favorite complex dishes. This allows them to save their money for other things.

2. Related Work

Artificial intelligence (AI) is “the theory and development of computer systems able to perform tasks that normally require human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages.” [1] AI has become one of the main focuses of computer science and software engineering research in the present day and keeps advancing everyday to develop the capabilities to perform tasks that were never thought possible. This chapter will introduce the foundation of research focused on the development of the AI-driven food recognition and recipe generation application created in this project.

2.1 AI-Based Image Recognition

“Computer vision is a field of artificial intelligence (AI) that enables computers and systems to derive meaningful information from digital images, videos, and other visual inputs.” [2] One application of computer vision comes in the form of AI-based image recognition. Image recognition allows software to correctly identify anything in digital images. There are two approaches when it comes to the development of image recognition software. “The conventional computer vision approach is a sequence of image filtering segmentation, feature extraction, and rule-based classification.” [3] However, developing such a pipeline requires a lot of time, expertise, and manual tweaking. [3] The other approach is the use of machine and deep learning in developing image recognition software. Rather than using a combination of all computer vision applications, this approach uses algorithms to train image recognition models based on large, categorized datasets of images. This approach doesn’t require as expensive hardware and has the potential to provide better performance [3]. It is also much simpler to use, requiring coding knowledge rather than expertise in computer vision. There are also many model making/training tools openly available to the public, such as Amazon Rekognition, that streamline the process for the user.

2.1.1 Amazon Rekognition:

Rekognition is an AWS service/tool that can be used "to easily add image and video analysis to AWS applications." [4] Within Rekognition, Amazon provides a large pretrained model that can recognize a multitude of objects. As can be seen in Figure 1, Amazon's pretrained model is able to recognize everything present in the image including each car's wheels, the person riding the skateboard, and the building on the left. While this model is very powerful, Rekognition also provides the user with the capabilities to train their own finely-tuned models using Amazon's training algorithms. Using Rekognition's Custom Labels, the user can upload their own refined dataset to train their own model.

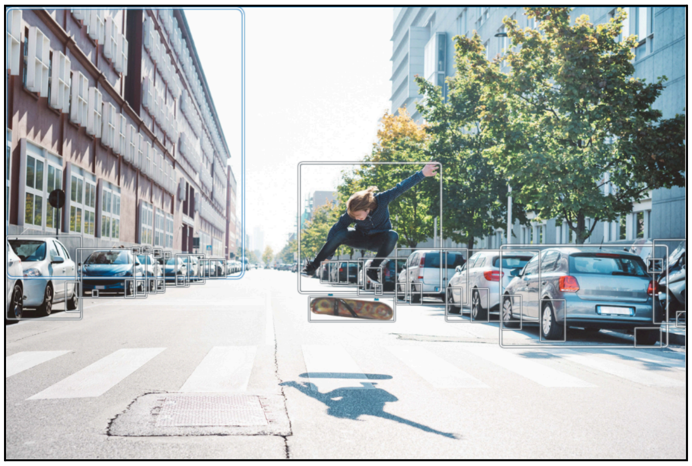


Figure 1: Capabilities of Pretrained Rekognition

To get started with using Custom Labels, first, one has to create an S3 bucket on their AWS account. Once the S3 bucket is created, the user can create a project in the Rekognition Custom Labels console. This project will store everything the user needs to train their model including their dataset of images they wish to recognize and the model itself. After the project has been created, the user will be brought to the project's home page. Within the homepage, the user is able to begin uploading their data and training their model.

The first step to training a model is creating the dataset the model will be trained off of. When creating a dataset, Rekognition allows the user to either upload one large dataset that will be used for training and testing or two separate datasets for each purpose. These datasets can either be imported from the user's PC or uploaded to and imported from the user's S3 bucket. Once a dataset is uploaded, the user can then generate labels to describe what is found in each image. Labels can be generated in two ways. The first involves the user organizing their images into folders for each object they would like to recognize and generating the labels automatically based on each folder's name. This approach is best for images with one object in each photo. The second allows the user to upload a folder of uncategorized images that they will manually label. This approach is best if the user is using images that include multiple objects.

After the dataset is uploaded and labeled, the model can finally be trained. Training models in Rekognition is as easy as a click of a button. If the user uploaded two separate datasets for training and testing, the training will begin immediately, but if the user is using one large dataset for both, Rekognition will split the data using 80% for training and 20% for testing then begin training. The speed of training models in Rekognition is efficient, but varies based on the size of the dataset and how many labels the user wishes to recognize.

Finally, once the model is trained, the user can evaluate it to see if it requires more training. When evaluating a model to see if it is ready for use, Rekognition provides several useful metrics to see how the model is performing. The precision of a model "is the fraction of correct predictions over all model predictions at the assumed threshold for an individual label."

[5] A model with high precision is cautious in making predictions and will only identify an object in a photo if it is sure it knows what it is, resulting in fewer false positives. [6] The recall of a model “is the fraction of your test set labels that were predicted correctly above the assumed threshold.” [5] A model with high recall is able to identify each custom label correctly when it's present in the testing dataset of images. [5] Each of these values is a fraction from 0 to 1.0 so a well trained model will have close to a 1.0 for both evaluation metrics. Once the user is happy with their model, they can deploy it using Python or console commands and the model’s Amazon Resource Name (ARN).

2.2 Amazon Web Services

“Amazon Web Services (AWS) is the world’s most comprehensive and broadly adopted cloud.” [7] AWS offers a wide range of cloud-based products “from infrastructure technologies like compute, storage, and databases–to emerging technologies, such as machine learning and artificial intelligence.” [7] These products/tools are easily accessible within the AWS system and have a wide range of services that can be used to support a complete full stack web development cycle and to create innovative products. Some important AWS services for this project include Amazon S3, DynamoDB, Amazon Amplify, and Amazon Rekognition as previously discussed.

2.2.1 Amazon S3

Amazon S3 is the cloud storage that will be heavily used to link the different AWS services together and share the files between them. “Amazon Simple Storage Service (Amazon S3) is an object storage service offering industry-leading scalability, data availability, security, and performance. It includes cost-effective storage classes and easy-to-use management features that can optimize costs, organize data, and configure fine-tuned access controls to meet specific business, organizational, and compliance requirements.” [8] Amazon S3 also provides a fast and accessible Python SDK that is easy to use and can be used by other Amazon services to access any related files. The code snippets below are examples of how a user can upload and download files from S3.

<u>Uploading a file:</u>

```
import boto3
client = boto3.client('s3')
with open("mnist.py", 'rb') as file:
    file_contents = file.read()
response = client.put_object(
    Body=file_contents,
    Bucket='examplebucket-8232936',
    Key='Media/mnist.py',
)
```

<u>Downloading the file:</u>

```
response = client.download_file(
    Bucket='examplebucket-8232936',
    Key='Media/mnist.py',
    Filename=r'A:/algorithm/DishRecognizerSP/python.py'
)
```

2.2.2 DynamoDB

DynamoDB is a serverless, NoSQL database that is highly scalable and can be used to store information. DynamoDB provides a simple Python API that allows data to be fetched from and sent to the database by the website backend. It is also compatible with Amazon S3’s cloud storage. The code snippet below uses the boto3 package to call the DynamoDB API using Python.

<u>Return the list of tables:</u>

```
import boto3
from botocore.exceptions import NoCredentialsError
try:
    # Initialize the DynamoDB client
    client = boto3.client('dynamodb', region_name='us-east-1')
    # List tables
    response = client.list_tables(Limit=10)
    # Print the list of table names
    print("Table Names:")
    for table_name in response['TableNames']:
        print(table_name)

except NoCredentialsError:
    print("AWS credentials not found. Please configure your credentials.")
```

2.2.3 Amazon Amplify

Amazon Amplify is used to handle the app from where the model would be accessible to users. Amplify provides the capabilities to host cloud-based web pages on Amazon servers, that is also compatible with other Amazon services that are used for a full-stack web development setup. The front end of the web app can be easily connected through Github and APIs and authentication can be handled by using other Amazon services (Lambda, API Gateway) that Amplify manages in the background. For the backend, Amplify will also use S3 and DynamoDB as primary storage tools. Amplify is also cost-efficient, as it allows the user to run an app for free for 12 months and then only pay for the time that the server is run.

Amplify uses Lambda functions, which are individual components that are designed to do one designated task and can be highly scalable. Every function can be custom-adjusted to use as much or as little memory as it needs. Lambda functions can be particularly useful when working with big chunks of data, small user requests, and model training and deployment. It matches the structure of this project since it will be working with big chunks of data, small user requests, and model training. So customization of the API will allow you to not pay any extra for the computational power that is not used.

2.3 Natural Language Processing

"A large language model (LLM) is a deep learning algorithm that can perform a variety of natural language processing (NLP) tasks." [9] LLMs are used to analyze and comprehend one's speech in order to generate a logical response to a user's query. This is achieved through training a complex model using enormous quantities of data and allowing it to infer relationships between words within the text. [10] With enough data and training, the model will then be able to predict what words would most likely follow a user's input to give the perfectly crafted response to any query the user might think of. One service that allows the use of their LLMs is OpenAI.

2.3.1 OpenAI

OpenAI is an AI research and development company that hosts some of the best large language models for text processing and generation. Through OpenAI's website, users can interact with their models online through services such as ChatGPT and DALL·E. Registered users are also able to use OpenAI's API in order to incorporate their models into their own applications. In order to use the API in Python code, the user must install and import the openai library. This library provides all of the necessary functions and capabilities for communicating with the API to take advantage of their models. Once installed and imported into the user's project, they can call the API in as little as 4 lines of code.

Using gpt-3.5-turbo in Python [6]:

```
import openai
# Your OpenAI account's secret key.
openai.api_key = 'YOUR_API_KEY'
# Prompt for the model.
prompt = 'Hello, how are you today?"
completion = openai.ChatCompletion.create(
                                    model='gpt-3.5-turbo',
                                    messages=[{'role':'user',
                                              'content':prompt}]
print(completion.choices[0].message.content)
```

In order to use the API in their programs, the user must first generate a secret API key on their OpenAI account that will be used to grant them access. This key is used to set the value of openai.api_key and tracks which account is using the service so that they can make sure the user is authorized to call the API. Once the API key is set, the user can then begin sending prompts to the API using the openai.ChatCompletion.create function. This function takes in the model the user wishes to use and multiple messages/prompts the model must respond to as parameters. Then, it returns the model's responses. For chat completion models, the user is able to choose from the GPT's as well as a few other models that are hosted on OpenAI. In the coding snippet above, the 'gpt-3.5-turbo' model is used because it is the best and cheapest model available on the free tier of ChatGPT. For messages, the user must include the role and the content of the message. The role of the message determines how the model will react to it. For example, if the role of the message is "user", the model will respond to the user's query like they are using ChatGPT, but if the role of the message is "system", it will determine the behavior of the model. The message content is the actual question of instruction the user is asking the model. An example of a message from the "user" role can be seen in the code snippet above where the user asks the model how they are. An example of a message from the "system" role can be "You are a terrible assistant". This will prompt the model to provide bad answers to the user's queries.

3. Design and Implementation

3.1 UI Design

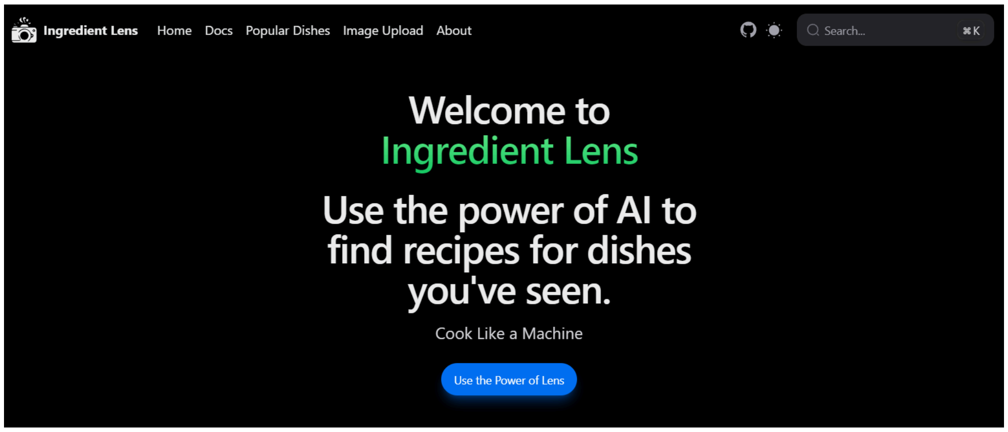
3.1.1 Desktop UI

The first impression of any website is important. For the final version of the app, we decided to use NextUI for the design of the application in order to achieve a more modernized, appealing design. The NextUI interface design framework allows users to incorporate a wide range of components that can transform the overall look of the website. It also gives users the liberty of combining many other tools like Tailwind, React, and Next.js. To accomplish this, all of the code is stored as Typescript files. This allows us to use pre-made methods in the NextUI liberty that look more eye-catching to the users. This also allows for a more in depth user experience, implementing features such as light and dark mode for the website that allows the user to change the website's theme with the click of a button.

In addition to NextUI, Next.js was also incorporated into the final iteration of the website. Next.js is a React framework that allows developers to create full-stack web applications by extending and integrating both React features and Javascript tooling. It supports dynamic HTML and CSS streaming, allowing the website to update itself in real-time as changes are made. Additionally, it features built-in optimizations for things like fonts, images, and scripts which automatically improve the website. There are also server-side features that improve the website's performance, like advanced route handling and data fetching. Utilizing NextUI and Next.js together allowed us to create a significantly more powerful and flexible website compared to the first draft which only used basic HTML and CSS.

Home Page

When a user first logs onto Ingredient Lens, they are greeted with our home page (Figure 5). This page welcomes the user to the website, contains a button that leads to the main Image Upload page of the website, and includes the navigation bar which the user can use to navigate to each page on the website. This navigation bar contains our logo, the name of the website, links to all website pages, a Github icon that leads to the project's Github, a button that switches between light and dark mode, and a search bar that currently has no functionality. This navigation bar is also present on every other page on the website. The homepage is coded in TypeScript and is composed of common HTML classes and NextUI classes.



Welcome to Ingredient Lens

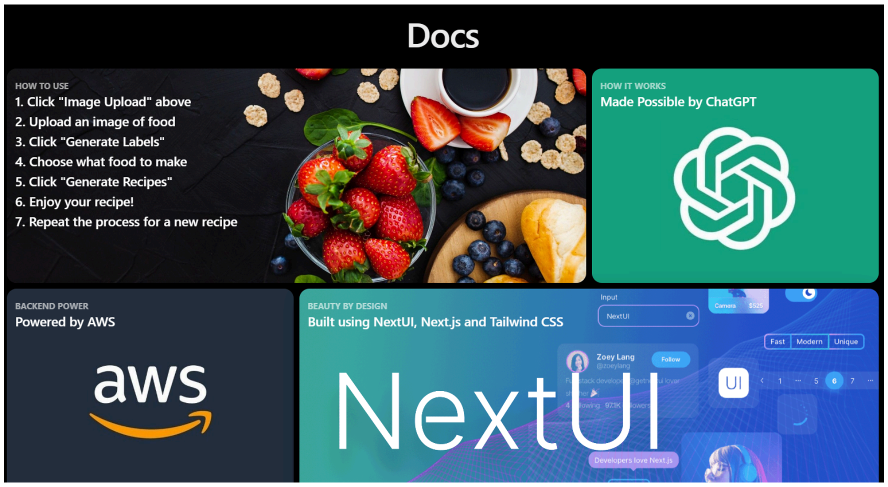
Use the power of AI to find recipes for dishes you've seen.

Cook Like a Machine

Figure 5: Ingredient Lens Homepage

Docs Page

When a user clicks the "Docs" button on the navbar, it brings them to the Docs page of the website (Figure 6). This page is composed of NextUI Cards that tell users about how Ingredient Lens operates. Each Card has a link, a heading, and subtext if needed. These cards are assembled in a visually appealing grid format. When clicked, each card will lead the user to the webpage of the tool used. The Docs page is also coded in TypeScript.



Docs

HOW TO USE

1. Click "Image Upload" above
2. Upload an image of food
3. Click "Generate Labels"
4. Choose what food to make
5. Click "Generate Recipes"
6. Enjoy your recipe!
7. Repeat the process for a new recipe

HOW IT WORKS

Made Possible by ChatGPT

BACKEND POWER

Powered by AWS

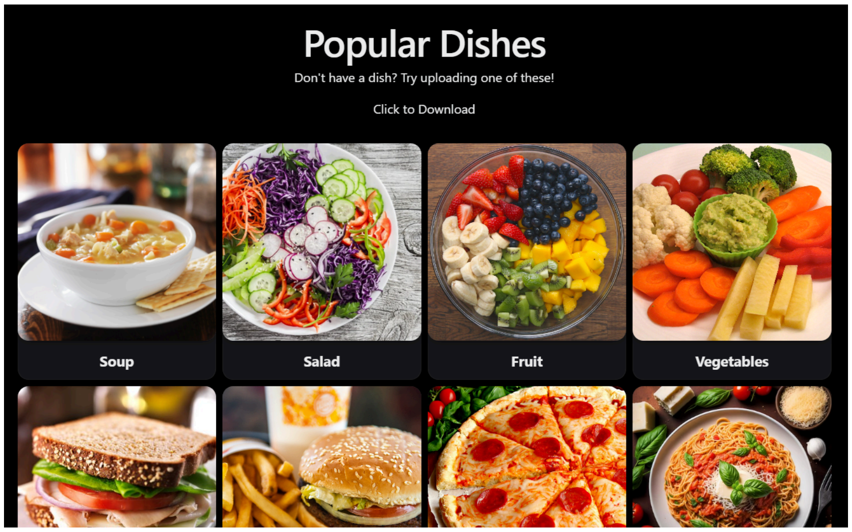
BEAUTY BY DESIGN

Built using NextUI, Next.js and Tailwind CSS

Figure 6: Docs Page

Popular Dishes Page

When a user clicks "Popular Dishes" on the navbar, it leads them to the Popular Dishes page (Figure 7). The Popular Dishes page lays out several example images that the user can download and upload to the Image Upload page to generate recipes. This page is composed of a list of NextUI Cards each with an image and a CardFooter explaining what the picture is of. When an image is clicked, it is downloaded to the user's computer to be available to upload to the Image Upload page. The Popular Dishes page is also coded in TypeScript.



Popular Dishes

Don't have a dish? Try uploading one of these!

Click to Download

Figure 7: Popular Dishes Page

Image Upload Page

When a user clicks "Image Upload" on the navbar, it leads them to the Image Upload page. When a user first enters the Image Upload page, it is pretty barebones (Figure 8). At first it is only composed of two NextUI Cards, one with file upload and generate label buttons and one with a line of text stating that it is where the outputted recipe will appear.

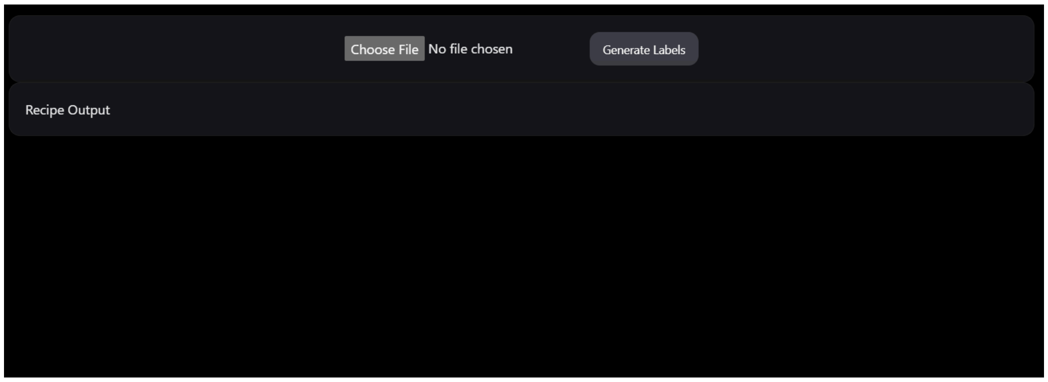


Figure 8: Initial Image Upload Page

Clicking the "Choose File" button opens up the file directory on the user's PC and allows them to choose an image to upload. Once a file is chosen, the image appears in an NextUI Image class of fixed size below the bar card (Figure 9).

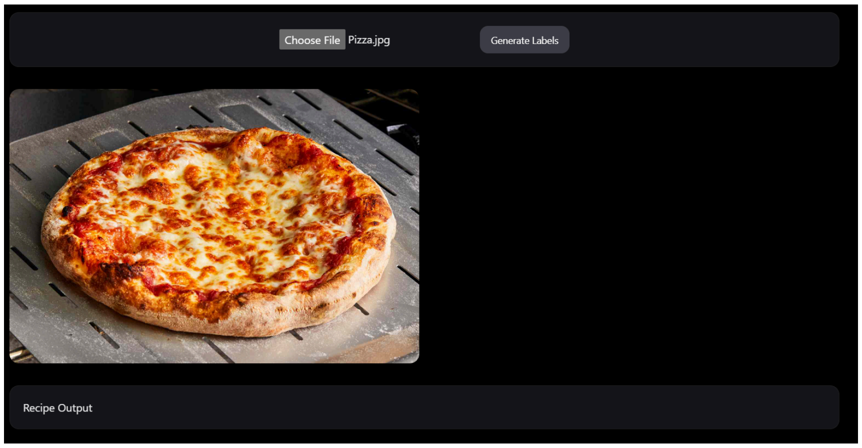


Figure 9: Website View When an Image is Uploaded

Once an image is uploaded, the user can click the "Generate Labels" button to send the image to our API Gateway to trigger the S3 and Amazon Rekognition Lambda functions. The labels gathered from the Rekognition Lambda will then be listed next to the image in a NextUI Listbox and clickable by the user. Once a label is clicked, it turns green and the "Generate Recipe" button will appear allowing the user to start the recipe generation process (Figure 10).

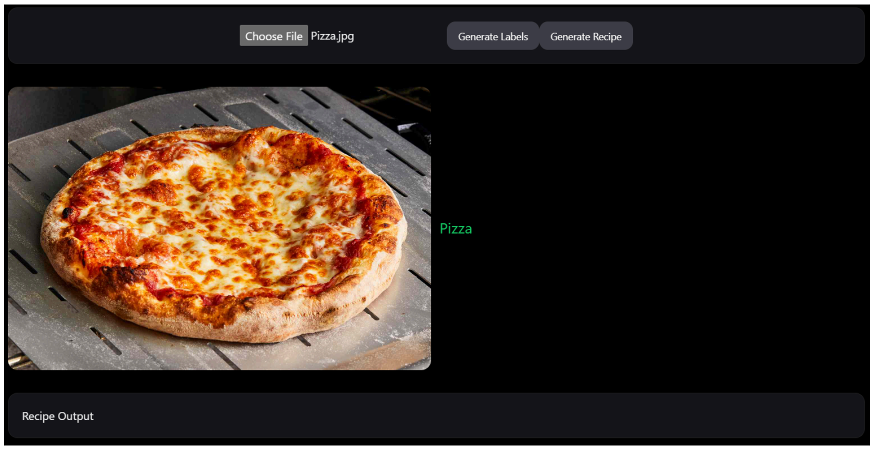
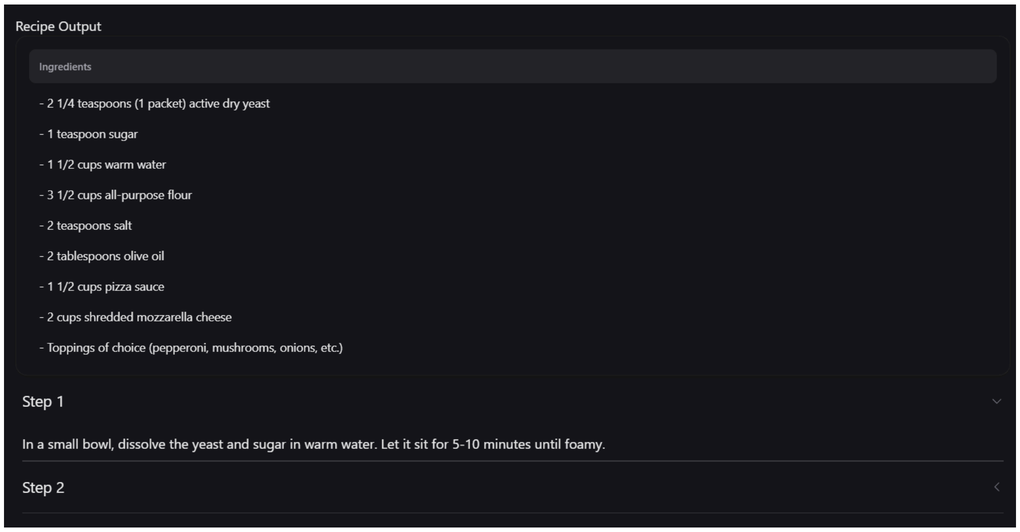


Figure 10: Website View after Label Generation and Selection

When the user clicks the "Generate Recipe" button, the website will call the OpenAI Lambda function directly and the OpenAI API will begin generating the recipe. Since the OpenAI API is sometimes slow, a loading bar will appear in the Recipe Output Card signifying that it is working (Figure 11).

Figure 11: Loading Bar

Once the recipe is generated, the loading bar disappears and the Recipe Output Card is filled with the ingredients and steps to make the dish (Figure 12). Ingredients are displayed in a NextUI Table and Steps are displayed in a NextUI Accordion. Ingredients are automatically viewable, but the user has to click each fold/step in the Accordion to view the contents of each step.


Recipe Output

Ingredients

- 2 1/4 teaspoons (1 packet) active dry yeast
- 1 teaspoon sugar
- 1 1/2 cups warm water
- 3 1/2 cups all-purpose flour
- 2 teaspoons salt
- 2 tablespoons olive oil
- 1 1/2 cups pizza sauce
- 2 cups shredded mozzarella cheese
- Toppings of choice (pepperoni, mushrooms, onions, etc.)

Step 1

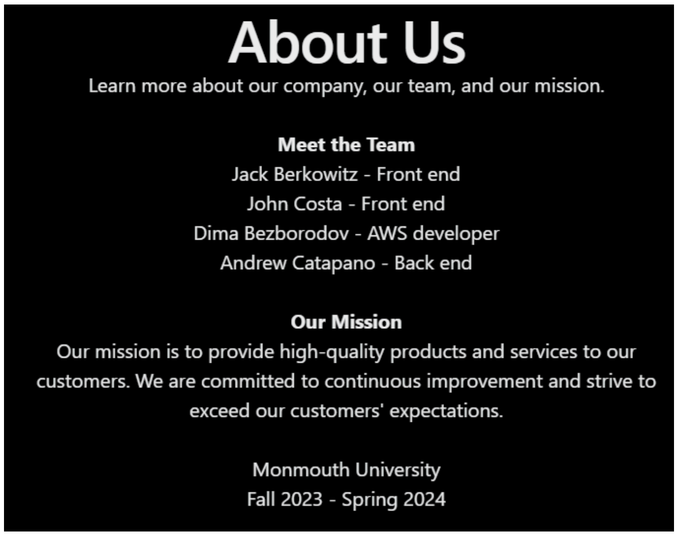
In a small bowl, dissolve the yeast and sugar in warm water. Let it sit for 5-10 minutes until foamy.

Step 2

Figure 12: Recipe View on Website

About Page

When a clicks “About” on the navbar, they are led to the About page (Figure 13). This page is very simple and only contains some basic information about our team such as our names and project. This page is the only webpage that doesn’t include any NextUI elements and is just coded in using basic HTML classes with Typescript.


About Us

Learn more about our company, our team, and our mission.

Meet the Team

Jack Berkowitz - Front end

John Costa - Front end

Dima Bezborodov - AWS developer

Andrew Catapano - Back end

Our Mission

Our mission is to provide high-quality products and services to our customers. We are committed to continuous improvement and strive to exceed our customers' expectations.

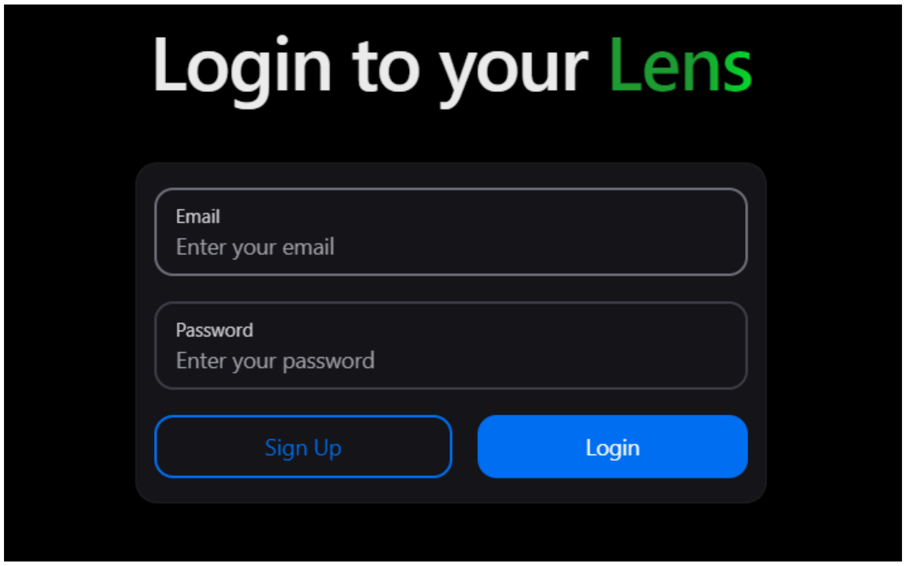
Monmouth University

Fall 2023 - Spring 2024

Figure 13: About Page

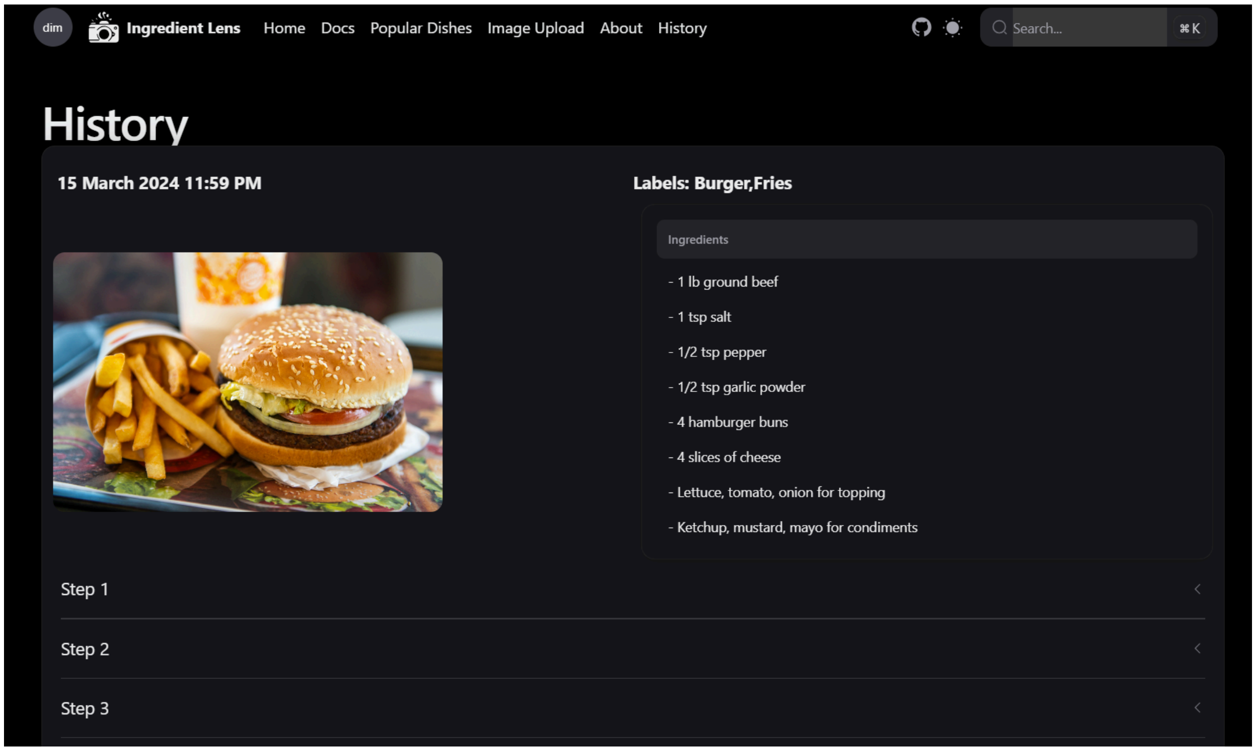
Login Page

The login page allows the user to sign up for IngredientLens and create an account to keep their information about dietary needs allergies and history of generated recipes. For now, we didn't focus on the specific user information, so the user just needs to input an email, and password which are stored in the database in hashed secured format.



History Page

The history page provides the user with a storage of their past generated recipes that they can revisit. It shows the date of the generated history item, the image of the uploaded image and labels that were assigned to the image. Also provides the generated step-by-step process of the recipe of the selected dish, as well as the list of ingredients required for that dish.



History

15 March 2024 11:59 PM

Labels: Burger,Fries

Ingredients

- 1 lb ground beef
- 1 tsp salt
- 1/2 tsp pepper
- 1/2 tsp garlic powder
- 4 hamburger buns
- 4 slices of cheese
- Lettuce, tomato, onion for topping
- Ketchup, mustard, mayo for condiments

Step 1

Step 2

Step 3

User Page

The user page holds important information about the user, where they can update the information about their dietary needs and allergies. This page can also be used in order to update user's personal information, such as name, email, and passwords.

3.1.2 Mobile UI

The first impression of any mobile application is important. For the final version of the mobile app, we decided to use React Native + Expo in order to achieve an appealing design. React Native and Expo come with a variety of components suitable for designing a mobile interface for both iOS and Android. The use of React Native also allowed us to easily connect our mobile frontend to our preexisting AWS backend. All of the code for the frontend is stored in either TypeScript or JavaScript files. This allows us to use pre-made methods in React Native and Expo libraries that are visually appealing and highly customizable. This also allows for a more in depth user experience enabling the use of navigators to seamlessly transition between pages.

Navigation:

For navigation throughout our app, we used a Drawer Navigator from the react-navigation library. A drawer navigator is used to render a navigation drawer on the side of the screen that can be opened and closed via gestures (swiping). Drawers are also highly customizable allowing for a more modern navigation design.

Pages:

Our mobile app contains all of the pages found on the website adapted for use on a mobile screen.

3.2 Front and Backend Design

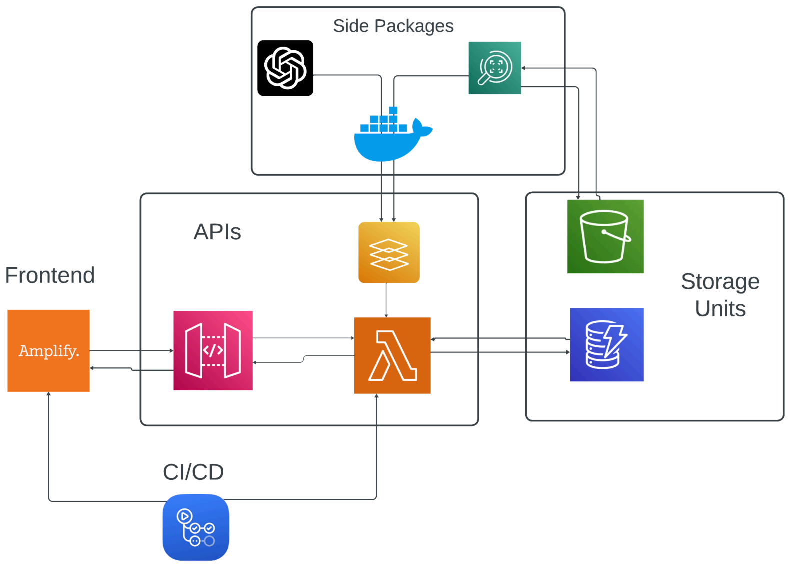


Figure 14: Workflow Chart

The workflow of this project can be broken down into several sections. That includes **Frontend,** which holds the UI/UX design that is presented to the user and runs the web server.

APIs that serve as a connection between frontend, side packages, and backend storage units.

Storage Units, that hold important files and information that is used in the project.

Side Packages, which are the outside software and tools that the project is using.

Continuous Integration/Continuous Delivery (CI/CD), which allows quick changes to the project and in-place updates.

In this section, there will be a detailed implementation of the above workflow, the challenges that were faced, and their solutions. Some of those approaches might not be optimal or may require changes in the future. It will require proper maintenance and even though the services that are in use are highly maintained, there might be better, more cost-efficient solutions.

3.2.1 Desktop Frontend

AWS Amplify allows users to run any type of React project on their Amazon server. By creating a simple Node JS project that contains CreateRoot functions that are fetched from React, Amplify is able to identify that as the root of the project, meaning that when the user goes to the general Amplify link, that will be the page shown. React supports various bootstrap options that will be used to create a UI. That approach allows users to apply all of the Object Oriented Programming requirements, which are the main traits of the React application.

In addition, Amplify allows direct connection with GitHub, which allows for the CI/CD approach, as all of the changes that will be made will automatically update the content of the website. That only applies to the front end, and there are other options to implement CI/CD that use GitActions. In addition, with Amplify, there is an easy way to add custom domains to the project that will be used in the future.

3.2.2 Mobile Frontend

React Native is a JavaScript library used to build user interfaces (UIs) for native apps. It can be used to create cross-platform applications that can be used on both Android and iOS. It also provides developers with all of the tools needed to create complex, responsive UIs that can interface with other code and APIs and enables fast iteration to see and test code changes in real-time.

For our mobile app, we used React Native for our frontend design because our website was already a React application. This allowed us to reuse most of the code to mimic the website's functionality in a mobile setting. Along with React, we used Expo. Expo is also an open-source platform for making universal native apps for Android, iOS, and the web with JavaScript and React. It contains libraries of components that can be added to React Native apps as well as enables the use of Expo Go for real-time app testing. Expo also contained many useful libraries to replace the libraries used on the website that were incompatible with React Native such as the crypto module.

From our React Native + Expo app, we were able to connect to our AWS backend to provide all of the functionality present on the website. The only main difference between the two is the UI design and different libraries to access a user's files and provide some backend functionality.

3.2.3 APIs

APIs are what allow the front and backend to communicate and react to the user's actions by responding with corresponding information or running processes. The initial call of the API happens with the front end, which then receives the information in JSON format and does some action that updates the webpage for the user. For this project, two APIs were used, one custom AWS API Gateway and the OpenAI API.

AWS API Gateways allow users to call Lambda functions from the front end using simple HTTP protocol. In this case, the trigger for the functions is the request that was made to the Gateway. Then, the Lambda function is executed. Our API Gateway handles four POST requests each corresponding to a specific Lambda function, “dynamo-get”, “dynamo-put”, “s3-upload”, and “rekognition”. Once a specific POST request is sent to the Gateway, it triggers the corresponding Lambda to return a response.

The OpenAI API allows users to call OpenAI’s LLMs like gpt-3.5 to return a response to a query to their own programs. The OpenAI API is used within the ChatGPT Lambda function to return a recipe when queried to create one for the food that is recognized. This Lambda function is the only one that is triggered directly rather than through our Gateway because there needed to be a longer timeout window than the max for the Gateway since the OpenAI API takes longer than 29 seconds to return a response.

Lambda functions can be written in any of the supported programming languages. For simplicity, the main language used will be Python, version 3.8. When uploading the function, the language, architecture, and function handler must be specified. That is integrated with GitHub Actions which allows users to push any folder that has Python files. Members of our team can easily push functions by specifying a path to the folder of files, function name, and path to the starting function. This way it’s quick and easy to make updates or add new functions since every part of the project is available on GitHub.

Furthermore, if a function requires any of the additional packages that need to be installed, a Lambda layer can be connected to the functions. Layers serve as the functional code that is used in different parts of the different functions. This lessens repeated code, as any Lambda function can inherit code from a layer.

3.2.4 Storage Units

Storage units store the main information about the users and files that are used for machine learning, specifically in Image Recognition. Those storage units can be accessible from the Lambda functions. DynamoDB serves the purpose of the database:

email (primary key, string) - Used as a unique ID of a user, which is a user's email address

Password (string) - Protected hashed password of the user

diet (string) - Used to keep track of the user's dietary restrictions

allergies (string array) - Used to keep track of the user's allergies that are selected from a middle-size list of common allergies.

history ([historyElement{date: string, ImageHash: string, labels, string array recipe: {ingredients: string array, steps: string array}} - holds the individual history of the generated recipes by the user, which is in JSON format. *Date* is the date and time when the recipe was generated. *ImageHash* holds the hash of the image that is stored in the s3 bucket. *Labels* are used to store labels that were assigned by the model to the image. The *recipe* stores a recipe on the selected label. Where *ingredients* are the set of ingredients generated and *steps* is a set of steps to prepare the dish.

Our current S3 bucket is used to store user uploaded images to potentially use them for model training, site statistics, and recipe storage in the future. The S3 path is assigned depending on the image hash and user login. The easy way to access any image is by following the path: *general-bucket/<user-email-login>/<imageHash>.*

3.2.5 Side Packages

As mentioned before, side packages and environment setup for Lambda functions are done through the Lambda layers. Layers allow any package to be used in functions that inherit from that layer.

The Docker container is used to set up all of the packages in one layer. In order to use a Docker container, first, the Docker Image must be initialized. This image contains the information on what type of packages and software must be initialized and which commands it needs to perform. In a way, it can be explained as a very small computer that has a very specific environment on it. In our case, it uses a Python Base image that is able to run Python code, then the image looks at the requirements.txt file and runs the pip installment for each of the requirements, which uploads all of the packages that will be used in the layer. Then, all of the modules are zipped into one folder that will be used to upload to AWS. After the image is created, the container starts its process by running it, executing all of the above requirements. After that, the zip file that was created in the Docker container is fetched and uploaded to the AWS server. This process is automated and can be performed through git actions, where members of our team can just include layer name, description, and path to the requirements.txt file. It will upload a created layer to the AWS server which is accessible through Lambda functions.

3.2.6 CI/CD

With CI/CD, we want to make sure that if there are any changes that need to occur or any bugs that need to be fixed, it can be done easily without issues. This way, when someone wants to commit changes to the project, it will need to pass several checks before being approved.

Some of those checks are programmatic, which can be updated with the Lambda functions, or human checks, where at least 2 team members need to approve the code to be integrated into the project. This makes the development easy to control and very fast paced, because all of those changes are happening in place. When changes are made, the Amplify server automatically looks at the gitHub updates and can update the website on the spot.

With git actions, developers can easily automate processes that are repeated on a daily basis such as creating Lambda functions, uploading side software, or cosmetic changes to the UI.

3.3 Implementation

3.3.1 Layers

Layers are implemented by using the Dockerfile along with git actions to dynamically add and update the packages that the project uses in lambda functions. First, when git actions are run, it sets up a git virtual machine, where it runs all of the code that allows creating package zip file and uploads it to AWS Lambda server:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2


      - name: Set up Docker
        run: |
          docker build --build-arg REQUIREMENTS_PATH=${{ github.event.inputs.path_to_req }} -t
lambda-layer .
          docker run --name lambda-layer-container -v ${{ github.workspace }}:/app lambda-layer
          docker cp lambda-layer-container:/app/layer.zip .
          docker stop lambda-layer-container
          docker rm lambda-layer-container
          docker rm --force lambda-layer
        working-directory: ${{ github.workspace }}
```

The Docker job is responsible for the creation of the docker container that will download all of the required packages. docker build allows us to build an image of the container, docker run runs the container and executes the code inside of it, and docker cp is responsible for copying the zip file from the docker container to the local virtual machine run by git. The remaining docker stop and docker rm commands clean the workspace because at this point docker finished the task.

The content of the Docker file is responsible for creating the packages and zipping them into one file, the comments in the code snippet explain in detail how this container works:

```
ARG REQUIREMENTS_PATH

# Use the official Python runtime as the base image
FROM python:3.8-slim-buster

# Set the working directory in the container
WORKDIR /app
# Install any necessary dependencies
RUN apt-get update && \
    apt-get install -y zip && \
    rm -rf /var/lib/apt/lists/*

# Copy the requirements file to the working directory
COPY $REQUIREMENTS_PATH .

# Install the Python packages listed in requirements.txt
RUN pip install -r requirements.txt -t /opt/python/

# Set the CMD to zip the installed packages into a layer
ENTRYPOINT ["/bin/sh", "-c", "cd /opt && zip -r9 /app/layer.zip ."]
```

$REQUIREMENTS_PATH variable is passed when creating the docker image, and is referencing the path of the file with requirements in the GitHub repository. After the zip file with the modules is created, the git virtual machine needs to push those packages into the layer on the AWS server. The code snippet below pushes that file into the layer:

```
- name: Create AWS Lambda Layer
        run: |
          aws lambda publish-layer-version \
            --layer-name ${{ github.event.inputs.layer_name }} \
            --description "${{ github.event.inputs.description }}" \
            --compatible-runtimes python3.8 \
            --license-info "MIT" \
            --zip-file fileb://layer.zip
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          AWS_EC2_METADATA_DISABLED: true
          AWS_DEFAULT_REGION: us-east-1
```

This code sets up the environment of AWS, where it fetches the access keys from the GitHub secrets. After the region is set, the aws lambda publish-layer-version command publishes the layer to aws, specifying the name of the layer, its description as well and the zip file that was created by the docker container.

After all of those actions, the layer can be attached to any of the lambda functions that use third-party packages. Next section lists some examples of the lambda functions that will be using outside packages, which will be used as APIs for the project's front and back end.

3.3.2 Lambda Functions

(1) Amazon Rekognition

The Rekognition Lambda function takes the image that was uploaded and passes it to Rekognition which returns the labels it recognized in the photo. Once called by the image upload, the function receives the image’s base64 data and decodes it into bytes. Once decoded, the image is passed to the detect_labels function(). After the labels recognized within the image are gathered from detect_labels(), the ‘Name’ and the ‘Confidence’ for each label are extracted and added to a json object which is appended to the *handled_response* array which is returned to the webpage as in json format.

<u>Rekognition Lambda Code:</u>

```
import boto3
import urllib.request
import urllib.parse
import urllib.error
import base64

client = boto3.client(‘rekognition’)

def lambda_handler(event, context):
    '''Demonstrates trigger that uses Rekognition APIs to detect labels in a
    base64 image.
    '''

    image_bytes = base64.b64decode(event["b_image"], validate=True)

    try:
        # Calls Rekognition DetectLabels API to detect labels in S3 object
        response = detect_labels(image_bytes)

        handled_response = []
        for elem in list(response):
            handled_response.append({
                'name' : elem['Name'],
                'confidence' : elem['Confidence']
            })

        # Print response to console.
        print(response)

        return {
            "labels": handled_response
        }

    except Exception as e:
        print(e)
        print("Error processing object.")
        raise e
```

The detect_labels() function uses the Rekognition client to detect the objects recognized in the uploaded photo. Within the request, the responses are limited to food and drink labels using the GENERAL_LABELS and IMAGE_PROPERTIES features. These features allow the user to limit pre-trained Rekognition's dataset of labels to only include labels useful to Ingredient Lens since a custom model hasn't been trained yet. Once these features are added, the user can add settings to the label detection such as "LabelExclusionFilters" and "LabelCategoryInclusionFilters". These are used to exclude certain labels from image detection and limit the categories of labels that are recognized. Since Ingredient Lens only deals with food and drinks, the "Food and Beverage" category is the only enabled one and the only things excluded from that category are the generic food labels like "Food", "Lunch", "Snack", etc.. Once the labels recognized within the image are generated, they are returned to be handled by the rest of the Lambda function.

detect_labels() Function:

```
def detect_labels(image):
    response = client.detect_labels(Image={'Bytes': image},
    MaxLabels=10,
    Features=["GENERAL_LABELS", "IMAGE_PROPERTIES"],
    Settings={"GeneralLabels": {"LabelExclusionFilters": ["Food", "Dinner", "Lunch", "Meat", "Meal",
"Snack", "Beverage"], "LabelCategoryInclusionFilters":["Food and Beverage"]},
     "ImageProperties": {"MaxDominantColors":10}}
    )
    return response["Labels"]
```

(2) OpenAI

The OpenAI Lambda function uses the output from the Amazon Rekognition Lambda to prompt the OpenAI API to get a recipe based on what dish was recognized. Once called, the function receives the label/food that was recognized, the user's allergies, and the user's dietary restrictions and inserts it into a prompt which we engineered to generate recipes with the same format no matter which food is inserted. Then, the function passes that prompt to the OpenAI API through the openai.ChatCompletion.create() function using the gpt-3.5-turbo model.

Once the recipe is generated it is passed through several regex expressions to capture the ingredients and recipe steps. In order to capture the ingredient, the expression used is '-\s\d?.*' which captures the ingredients listed because they always begin with a '-'. In order to capture the steps, there are several expressions used to capture the steps based on the output variation that is generated. With our current prompt, there are two main recipe variations that are generated. The first expression used is '\n\s+\d+\.(?!\sInstructions:)(?!\sDish\sName:\s)(?!\sIngredients:)(.+)' which captures every step in a singular string given a recipe that numbers each step and section of the recipe, "2. Ingredients", "3. Instructions", etc.. The second expression used is r'\n\d+\.(.+)\n' which captures every step in a singular string given a recipe that only numbers each step and not the recipe's sections. This second expression is only triggered if the first expression fails and captures nothing. Once the steps are captured in a singular string from either expression, the string is split using the re.split() function which splits the string into an array of each step contents. The pattern used to split the string is '\d+\.' which is a number with a period.

After all of the steps are gathered from the recipe, a for loop is used to iterate a step number and strip each step of any extra spaces that surround the text. Each step and the current step number are combined into a json object that is appended to the *handled_steps* array. After all steps are handled, the *handled_steps* array is combined with the *ingredients* array into a json object *recipe_data* which is returned to the webpage to be displayed.

In order for this function to run, an OpenAI API key was required. Since we are running our website code on AWS through our GitHub repository, we couldn't just upload the function with an exposed API key so we had to find another way to import the key securely. Within AWS Lambda functions, the user is able to set up environment variables that are exclusive to each function. This allowed us to set up an environment with the variable *access_key* which contained our API key. In order to import this information, we used the *os* Python library to access our key for use in the function.

OpenAI Lambda Function Code:

```
import openai
import os
import json
import re

def lambda_handler(event, context):
    # Gets access key from environment variables.
    openai.api_key = os.environ['access_key']

    print(event)
    print(event['body'])
    body_dict = json.loads(event['body'])
    label_value = body_dict.get('label')

    allergies = body_dict.get('allergies')
    diet = body_dict.get('diet')

    allergies_string = ""
    if allergies is not None:
        allergies_string = " without "
        for allergy in allergies:
            allergies_string += str(allergy)
            allergies_string += ", "

    diet_string = ""
    if diet is not None:
        diet_string = " " + str(diet) + " "

    prompt = "Generate a step-by-step recipe for " + diet_string + str(label_value) + allergies_string + """ in a
format suitable for display on a website. Include a list of ingredients and detailed instructions for each step.
Ensure the format is clear, consistent, and optimized for website display, with ingredients listed first
followed by sequential steps. Use the following format:
```

```
Recipe Prompt Format:

1. Dish Name: [Enter Dish Name]
2. Ingredients:
   - [Ingredient 1]
   - [Ingredient 2]
   - [Ingredient 3]
   - ...
3. Instructions:
   1. [Step 1]
   2. [Step 2]
   3. [Step 3]
   4. ..."""

    # Gets gpt-3.5-turbo's response to the prompt.
    completion = openai.ChatCompletion.create(model="gpt-3.5-turbo", messages=[{"role":"user",
"content":prompt}])
    recipe = completion.choices[0].message.content
    # Prints and returns the response in json format.
    print(recipe)

    # REGEX STUFF
    ingredient_pattern = r'-\s\d?.*'
    # Find all matches of ingredients in the recipe
    ingredients = re.findall(ingredient_pattern, recipe)
    # Display the captured ingredients
    for ingredient in ingredients:
        print(ingredient)

    step_pattern = r'\n\s+\d+\.(?!\sInstructions:)(?!\sDish\sName:\s)(?!\sIngredients:)(.+)'
    numbered_step_pattern = r'\n\d+\.(.+)\n'
    # Find all matches in the instruction text
    steps = re.findall(step_pattern, recipe, re.DOTALL)
    print(steps)

    if len(steps) == 0:
        steps = re.findall(numbered_step_pattern, recipe, re.DOTALL)
    print(steps)

    split_pattern = r'\d+\.'
    splitSteps = re.split(split_pattern,steps[0])
    print(splitSteps)
    steps = splitSteps

    handled_steps = []
    for step in steps:
        step_description = step.strip()

        """handled_steps.append({
            'number': step_number,
            'description': step_description,
            #'parts': step_parts
        })"""

        handled_steps.append(step_description)
        print(f"Step : {step_description}")
        print()

    recipe_data = {
        'ingredients': ingredients,
        'steps': handled_steps
    }

    response = {
        'statusCode': 200,
        'body': json.dumps(recipe_data)
    }
    return response
```

(3) S3

The S3 Lambda function connects the Amazon s3 file management system with the website. Every time the user uploads an image to the website and presses “Generate”. The website makes a call to that function and the image is stored in the database for future use in training:

```
import boto3
from botocore.exceptions import ClientError
import uuid
import base64

def upload_file(event, context):
    """Upload a file to an S3 bucket

    :param file_name: File to upload
    :param bucket: Bucket to upload to
    :param object_name: S3 object name. If not specified then file_name is used
    :return: True if file was uploaded, else False
    """
    # Create a random Id
    random_uuid = uuid.uuid4()

    # Decode the base64-encoded image data to bytes
    image_bytes = base64.b64decode(event["b_image"], validate=True)

    # Extract the first 8 characters from the UUID
    random_id = str(random_uuid)[:8]

    # Upload the file
    s3_client = boto3.client('s3')
    try:
        response = s3_client.put_object(
            Body=image_bytes,
            Bucket=event["bucket"],
            Key=event["key"],
        )
        return True
    except ClientError as e:
        return False
```

In short, the base64 image is passed into the function. After which the image is decoded to regular bytes and a random 8-digit id is generated to assign it to the image to distinguish between images in s3. Boto3 is an included library that any lambda function can use without a layer, and with the permission for lambda to call s3 the function is able to run serverless.

(4) DynamoDB

DynamoDB functions are there to connect to the database. For now, the functions of the database are separated into several functions of GET and PUT methods. However, once the project reaches new ground, the Lambda function can be abstracted using PartiQL. Which will allow us to do direct SQL queries to the database:

```python
import boto3
import hashlib
import json
def lambda_handler(event, context):
    """Function returns the bool value depending if the user is registered or not in the
    database given their login and password

    :param login: email of the user
    :param password: password of the user
    :returns: Boolean of whether the user was found or not
    """
    try:
        #connect to the database
        dynamodb = boto3.resource("dynamodb")
        table_name = "LoginInfo"
        table = dynamodb.Table(table_name)

        response = table.get_item(Key={'email':event['login']})

        sha256 = hashlib.sha256()
        sha256.update(event['password'].encode('utf-8'))
        hashed_password = sha256.hexdigest()

        if(hashed_password == response["Item"]['password']):
            status = True
        else:
            status = False
        resp = {
            'status' : status
        }
        return json.loads(json.dumps(resp))
    except Exception as e:
        data = {
            'status' : False,
            'error' : e
        }
        return json.loads(json.dumps(data))
```

The function first connects to the database and gets the row of the data if the user login exists. Then, the function processes the user entered password to hash value and compares it with the one on the database and if they match the return status is True. Otherwise the function returns the status of False in any other scenario.

PUT Method Code:

```
    try:
        dynamodb = boto3.resource("dynamodb")
        table_name = "LoginInfo"
        table = dynamodb.Table(table_name)

        sha256 = hashlib.sha256()
        sha256.update(event['password'].encode('utf-8'))
        hashed_password = sha256.hexdigest()

        response = table.put_item(Item={"email": event['login'], "password":hashed_password},
 ConditionExpression="attribute_not_exists(login)")

        return {'exist': False}
    except Exception as e:
        print(e)
        return {'exist': True,
                'error': e}
```

The function connects to the database and converts the passed password into a hash value for security reasons. Then, the function makes an attempt to put it into the database, but will only do so if entered email does not exist in a database. Otherwise, the return statement will be True, implying that there is a row with the given email in a database.

These functions are not in use in the current iteration of the website, but will be used next semester to implement user accounts.

(5) Update History

The function is called when the recipe is generated and if a user is signed in to their account. This means the information is not being stored if the user is in the guest mode. Data is passed in the JSON format required by the DynamoDB boto3 library:

```
dynamodb = boto3.resource("dynamodb")
table_name = "LoginInfo"
table = dynamodb.Table(table_name)
response = table.get_item(
    Key={
        'email': str(event['login']),
    },
    ConsistentRead=True
 )
item = response['Item']
if item:
# Step 2: Update the array in the item
array_field_name = 'history'  # Replace with the name of your array field
# Initialize the array if it doesn't exist
item[array_field_name] = item.get(array_field_name, [])
# Add a new dictionary to the array
item[array_field_name].append(event['historyItem'])
# Step 3: Save the modified item back to DynamoDB
update_response = table.update_item(
        Key={
            'email': event['login'],
        },
        UpdateExpression=f'SET {array_field_name} = :newArray',
        ExpressionAttributeValues={
            ':newArray': item[array_field_name],
        },
        ReturnValues='ALL_NEW',  # Change as needed
)
updated_item = update_response.get('Attributes')
return {'status' : True, 'updated_item': updated_item}
```

Function connects to the database and if the history array already exists, it appends the information to the existing array of history items and creates a new empty array if history is empty.

(6) GetItem

Is used to retrieve an item that is unique to the user, it uses the user’s email (primary key) in order get the required database row. Then function retrieves one out of the 4 items bounded to the user account (password, allergies, diet, history):

```
dynamodb = boto3.resource("dynamodb")
table_name = "LoginInfo"
table = dynamodb.Table(table_name)
response = table.get_item(
    Key={
        'email':str(event['login'])
    },
    ConsistentRead=True
)
item = response['Item']
attribute_value = item.get(event['item'])
return {'status' : True, 'item' : attribute_value}
```

(7) UpdateAllergy / UpdateDiet

This function is very similar to the UpdateHistory function, and does require in-depth explanation. The only difference is that the function contains the condition, as we also want availability to remove and add allergy:

```
if(event['action'] == 'delete'):
    item[array_field_name].remove(event['allergy'])
if(event['action'] == 'add'):
    item[array_field_name].append(event['allergy'])
```

Action parameter is passed into the JSON format, and action to the database are performed according to that parameter. If parameter is invalid or not specified, then no changes are made to the database.

(8) Fetch Image

This function allows to create the link that will point to the image on the S3 instead of downloading the image and storing it into the web server. This approach allows for easier and faster render of the pages that use a lot of pictures that need to be displayed. That is in the example of the history page:

```
    s3_client = boto3.client('s3')
    try:
        urls = []
        for path in event['images']:
            response = s3_client.generate_presigned_url(
                'get_object',
                Params={'Bucket': event["bucket"],
                        'Key': path},
                ExpiresIn=3600)
            urls.append(response)
        # Read the content of the response
        return {'status' : True, 'images' : urls}
```

This call takes all of the paths of the images from which the links must be created, the order is preserved and this way the history is still going to be ordered by the date of creation.

3.3.3 Mobile App Development

When designing the Ingredient Lens mobile application, the goal was to translate all the features and functionality of the website into a pocket-sized format. Having access to Ingredient Lens on the go could be very useful, for instance if one is deciding what foods to buy at the supermarket. It would be inconvenient if users needed to transfer photos taken with a phone over to a desktop computer before they could use the website.

The first question that needed to be answered was which platforms to support. Both iOS and Android are popular mobile operating systems. All members of the development team, however, use Apple devices. There was concern about not being able to properly test the app on Android. Luckily, there are tools that circumvent the issue and streamline the development process. To reach the most users, the decision was made to develop a cross-platform application.

One tool used early on was Expo Snack. Snack is a website that allows users to write code on the left-hand side of the screen and see their mobile app update on the right-hand side in real time. Users can import their GitHub repositories with the click of a button. It also has options to simulate both iOS and Android devices. This was very useful in the beginning stages, but as the codebase got larger, Snack encountered issues with not being able to import certain dependencies. Without them, the simulated app seen on Snack began to deviate from what the real app would look like on an actual device where everything is properly imported. Thus, the development team began to move away from Snack.

That is where Expo Go came in. Created by the same company that made Snack, Go takes a different approach to mobile app development. Go itself is a free mobile app which anyone can download. On a desktop, with one's workspace properly configured, a developer can run the code "**npx expo start**" to generate a QR code. Scanning this code with the same mobile device that Go is installed on links the workspace to the device, allowing Go to perform the same function as Snack. In other words, Go will open automatically and run the code for the mobile app, allowing developers to test their app on a real mobile device, with the app updating in real time as changes are made.

Initially, some problems were encountered trying to get Go properly set up. It seemingly refused to cooperate. Sometimes it would not load at all. Other times it would load, but touch input would be ignored. After some troubleshooting, the app began functioning as intended. Using Go greatly expedited development of the Ingredient Lens mobile app despite the initial hiccups. With that, the team was finally able to focus on the mobile app's features.

The app was coded using React.js. At first, the plan was to add buttons to the app's homepage which users would tap on to travel to other pages. It worked well enough, but turned out clunky and visually unappealing. So the buttons were scrapped and a Screen Navigator was implemented. It added a top navigation bar to every screen of the app. This method of navigation was easier to use, but lacking in customization options. The end result was very much function over form. It was an improvement for sure, but the team knew that more could be done.

The final version of the app uses a Drawer Navigator. This is another form of navigation menu that is much sleeker and less intrusive. Users open it by sliding their thumb from the left edge of the screen to the right. The menu will then slide out, revealing a list of all screens in the app. Those are Home, Login, Docs, Popular Dishes, Image Upload, and About. There is also a Sign Out button for users who are logged in. The Tell a Friend button was initially going to link to social media, but was later removed. Drawers are much more customizable, allowing the team to implement a white and green color scheme which falls more in line with Ingredient Lens branding. Each button also has a custom icon which corresponds to the purpose of each screen.

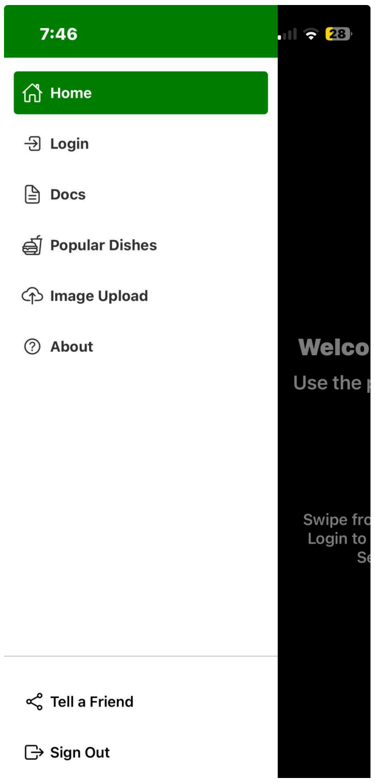


Figure 15: The Drawer Navigator menu

Lastly, each screen was implemented. Home looks almost identical to the website's homepage, with an additional tutorial explaining how to use the app. Login and Sign Out allow a user to login and sign out of their account. Docs has more information on how to use the app and how it was created. Popular Dishes features a selection of images of food which the user can tap to download. After saving an image to their device, the user can upload it on the Image Upload screen to get a recipe and list of ingredients. The About page gives some information about the development team and the project itself.

User Sessions:

On the mobile app, user sessions are managed through an AuthContext. This AuthContext stores information for use across all app screens. Within the AuthContext, a context is created using react createContext and an AuthProvider is created in order to make the context available to all screens. This AuthProvider is then used to wrap the whole App.

AuthContext.js:

```
import React, { createContext, useState, useEffect } from 'react';
import AsyncStorage from '@react-native-async-storage/async-storage';

// 1. Create Context
export const AuthContext = createContext();

// 2. Provider Component
export const AuthProvider = ({ children }) => {
  const [userToken, setUserToken] = useState(null);

  const login = (token) => {
    setUserToken(token);
    AsyncStorage.setItem('userToken', token);
    console.log(userToken);
    console.log("logged in");
  }

  const logout = () => {
    setUserToken(null);
    AsyncStorage.removeItem('userToken');
```

```
   console.log("logged out");
  }

  const isLoggedIn = async () => {
   try {
     let userToken = await AsyncStorage.getItem('userToken');
     setUserToken(userToken);
   } catch(e) {
     console.log('isLoggedIn error');
   }
  }

  useEffect(() => {
   isLoggedIn();
  }, []);

  return (
   <AuthContext.Provider value={{login, logout, userToken}}>
     {children}
   </AuthContext.Provider>
  );
};
```

App.js:

```
import { AuthProvider} from './src/AuthContext.js';

import AppNav from './src/Navigation/AppNav.js';

export default function App() {

  return (
   <AuthProvider>
     <AppNav />
   </AuthProvider>
  );

}
```

When a user logs in, a JWT is generated through the expo-jwt library, which is then stored in a useState in the AuthContext. In order to generate JWTs in React Native, we had to find an alternative library to the base jwt library since React Native doesn't come equipped with the crypto library. Along with being stored in the context, the token is also stored in React

Native's AsyncStorage which stores information in a cookie-like state for use everytime the user opens the app until they logout. When the token exists in AsyncStorage, the user gains access to a new navigation drawer that gives them access to their user settings and the sign out button. This token is also used to retrieve their dietary restrictions and allergies from the database for use in recipe generation.

4. Conclusion and Future Work

In this project, we created Ingredient Lens. Ingredient Lens is an AI-based application that allows the user to upload a picture of a dish. Then it recognizes the dish in the photo and outputs a recipe on how to make it. It was implemented on a website using a combination of AWS Services, React JS, the OpenAI API, Python code, NextUI, and Next.js. It was also implemented on mobile using React Native and Expo.

When creating Ingredient Lens, our vision was to create an application that would help users learn to cook dishes they have never cooked before, exposing them to a wide range of cooking skills and ingredients and increasing their overall knowledge in the kitchen. This vision was carried out by providing a simple way for users to generate simple recipes based on dishes they've seen rather than going through the struggle of finding a good one online.

In the future, we plan to expand the capabilities of Ingredient Lens by adding new features. Some of the additional improvements can be the custom-trained or fine-tuned model, which would have greater recognition accuracy. While a lot of effort was made to create a complete prototype of the mobile app, the final goal would be to deploy it on all mobile platforms for users to use and test the beta version of the project. In addition, we are looking forward to further personalize the user experience with potentially implementing the ranking system, where users can rank how easy or hard the recipe was and whether or not the dish is aligned to the user's needs. This approach might require finetuning and adding new features to the LLM model which we will look at in the future. Also, some of the information tied to the user's account can be more adjustable. Information such as the user's age, sex, and background can severely impact their preferences in recipes. It would be a great opportunity to explore the possibilities of using those parameters to impact the generated recipes.

References

[1] "Artificial Intelligence," Oxford English Dictionary, https://www.oed.com/dictionary/artificial-intelligence_n?tab=factsheet&tl=true (accessed Sep. 29, 2023).

[2] "What is Computer Vision?," IBM, https://www.ibm.com/topics/computer-vision (accessed Sep. 25, 2023).

[3] G. Boesch, "Image recognition: The basics and use cases (2023 guide)," viso.ai, https://viso.ai/computer-vision/image-recognition/ (accessed Sep. 25, 2023).

[4] "What is Amazon Rekognition?," Amazon, https://docs.aws.amazon.com/rekognition/latest/dg/what-is.html (accessed Sep. 24, 2023).

[5] "What is Amazon Rekognition Custom Labels?," Amazon, https://docs.aws.amazon.com/rekognition/latest/customlabels-dg/im-metrics-use.html (accessed Sep. 25, 2023).

[6] ChatGPT in Python for Beginners - Build A Chatbot. The AI Advantage, 2023.

[7] "What is AWS," Amazon, https://aws.amazon.com/what-is-aws/ (accessed Sep. 29, 2023).

[8] "Amazon S3", Amazon, https://aws.amazon.com/s3/ (accessed Sep. 23 2023)

[9] “What is a large language model?,” Elastic, https://www.elastic.co/what-is/large-language-models#how-do-large-language-models-work (accessed Sep. 29, 2023).

[10] M. Ruby, “How ChatGPT works: The Model Behind the Bot,” Medium, https://towardsdatascience.com/how-chatgpt-works-the-models-behind-the-bot-1ce5fca96286 (accessed Sep. 29, 2023).