Project Report: Magic Barber

COMS E6998 Spring 2022 - Cloud Computing and Big Data, Mentored by Yeqing Lin

Jianyang Duan
Columbia University
jd3794@columbia.edu

Jincheng Xu
Columbia University
jx2467@columbia.edu

Ruisi Wang
Columbia University
rw2902@columbia.edu

Yuerong Zhang
Columbia University
yz4143@columbia.edu

ABSTRACT

Magic Barber mainly solves the problem of what color to dye hair and where to do hair dyeing. People can get back generated dyed hair photos with recommended random hair colors by uploading their own hair photos to see the effect of hair dyeing in advance. They can also search inside Magic Barber for the nearby barber shops in the real world by entering their zip code.

Code Repo: https://github.com/jenniferduan45/Magic-Barber
Demo Video: https://youtu.be/t1Iv0SsUHFI

1 INTRODUCTION

Choosing a hair color to dye can be difficult because we cannot predict the effect of hair coloring in advance. Before we get our hair dyed in specific colors, we may want to know what these colors look like in our hair and if they are suitable. Therefore, we would like to help people who want hair dyeing to decide which color to choose.

It is also quite important for people to find the nearest barber shops around them after they decide what color to dye. Therefore, people can navigate through tons of real barber shops around them by simply entering their zip code in the search bar.

Our project aims to build a web application, Magic Barber, on helping people decide what color to dye and where to do hair dyeing. It will allow users to upload their own hair photos and get their photos back with recommended random hair colors dyed by a real-time, deep learning-based hair segmentation technique. Furthermore, users can type in their zip code to find nearby suitable barber shops.

2 ARCHITECTURE

Figure 1 shows the overall architecture of Magic Barber. The architecture flow is the following:

In the home page, the user can upload a photo to get dyed hair photos with different hair colors. Once the user clicks submit on the frontend, the /upload endpoint with PUT request in the API Gateway uploads the user's photo to the input S3 bucket and then triggers LF1. (For preparation, our deep learning model to perform hair segmentation is trained locally, stored in the model S3 bucket, and deployed to the SageMaker endpoint.) LF1 generates three photos with different hair colors by invoking the SageMaker endpoint with the user's photo and then stores them in the output S3 bucket.

In the recommendation page, LF2 accesses the dyed hair photos stored in the output S3 bucket and returns them to the API Gateway. These photos are displayed in the frontend for the user to view.

In the search page, when the user enters their zip code to search for the nearby barber shops, GET request is sent to the /search endpoint in the API Gateway to trigger LF3. LF3 uses this zip code to search barber shops in OpenSearch, and then uses the barber shops business ID(s) returned from OpenSearch to query complete barber shops data stored in DynamoDB. Barber shops data in DynamoDB and OpenSearch is pre-crawled using the Yelp API. After that, LF3 parses all detailed resulted barber shops data and sends them back to the API Gateway. Finally, these barber shops data are displayed in the frontend for the user to view.

3 API DESIGN

Magic Barber is designed to have three API methods: `PUT/upload`, `GET/results`, and `GET/search`.

3.1 PUT /upload

The /upload endpoint receives the PUT request with the image data in base64 encoding being the request body and stores image in the input S3 bucket.

Parameters: image file name, input S3 bucket name
Request Body: image data in base64 encoding
Response Body: N/A

3.2 GET /results

The /results endpoint receives the GET request and invokes LF2 to get the list of dyed hair photo URLs as response.

Parameters: N/A
Request Body: N/A
Response Body: a list of image URLs from the output S3 bucket in JSON format

3.3 GET /search

The /search endpoint receives the GET request with zip code being the query path parameter and invokes LF3 to get the nearby barber shops data as response.

Parameters: zip code

COMSE6998 CCBD, Spring, 2022
2022.

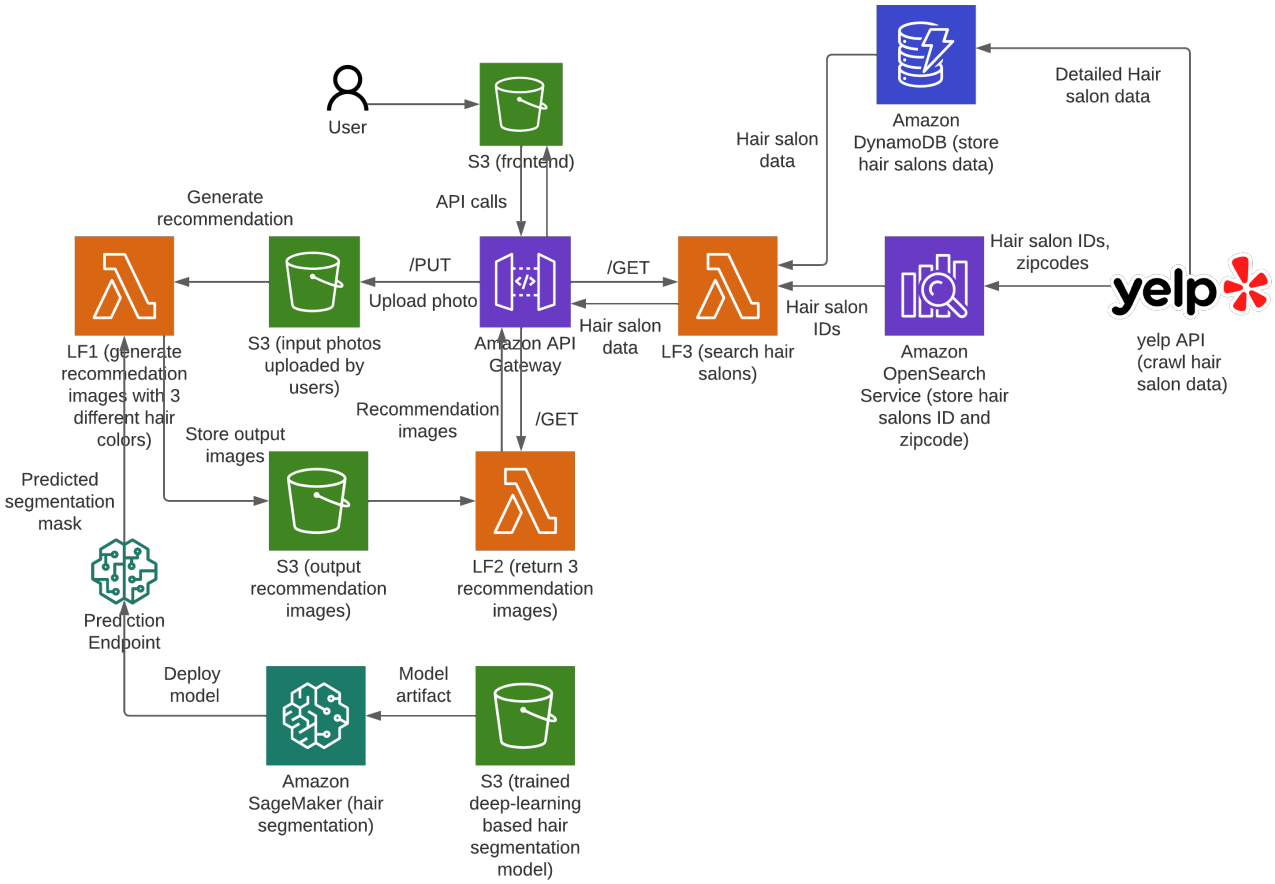

Figure 1: Architecture Diagram


```
{
    "BusinessId": "EyUbDeZLG3pnwUomOXINRQ",
    "BusinessName": "MASA.KANAI",
    "Address": "570 Columbus Ave, New York, NY, 10024",
    "PhoneNumber": "+19174092432",
    "Rating": "4.5",
    "Image": "https://s3-media4.fl.yelpcdn.com/bphoto/8beKwDYckV88B6aZEsK9dQ/o.jpg"
},
```


Figure 2: A sample barber shop data in GET /search response body

Request Body: N/A

Response Body: a list of nearby barber shops data in JSON format. Figure 2 shows one sample barber shop data in the list. More details about the data are illustrated in Section 4.2 and 4.3.

4 PROJECT DETAILS

The technical details of our project can be divided into the following parts: (1) SageMaker; (2) DynamoDB, OpenSearch, and Yelp API; (3) Lambda; (4) frontend.

4.1 SageMaker

We utilize the Pyramid Scene Parsing Network (PSPNet) [5], a state-of-the-art image segmentation algorithm, as the model backbone to segment the mask of hair. We choose Figaro1K [4] as the dataset, which contains 1050 hair images belonging to seven different hairstyles. We split it into training (60%), validation (20%), and test sets (20%). We evaluate our model performance by reporting the mean Intersection over Union (mIoU) between predicted masks and corresponding ground truth masks. Then we deploy our trained model [2], which is implemented in PyTorch, to an endpoint in SageMaker to do inference.

4.2 DynamoDB, OpenSearch, and Yelp API

We collect all barber shops that are located in Manhattan from the Yelp fusion API [1] using the request GET with the base URL: https://api. yelp.com/v3/businesses/search. Next, by running our Python script, we crawl over 300 barber shops data with the Yelp API. We then store them in a DynamoDB table. Each barber shop there has an unique business ID, address, an URL of a photo of the business, business name, number of reviews, phone number, rating, and zip code. In this table, the business ID is a primary key, and zip

business_id	address	image_url	name	number of reviews	phone	rating	zip code
string	string	string	string	int	string	decimal in range [1, 5]	string

Figure 3: Barber shops data stored in DynamoDB

code will be used as the searchable parameter later when users want to search for nearby barber shops. If any value except the business ID and zip code in barber shops data is null, we shows an 'N/A' for this value. Figure 3 shows the barber shops database schema stored in DynamoDB. To enable faster searching nearby barber shops by zip code, we also upload each barber shop's business ID and zip code to OpenSearch.

4.3 Lambda

We code three Lambda functions to connect all services to work together, including the API Gateway, SageMaker, the input and output S3 buckets, OpenSearch, and DynamoDB.

4.3.1 Generate Recommendations Lambda Function (LF1). Figure 4 shows the overall flow of LF1, which mainly generates dyed hair photos using the user's uploaded photo. For any new user's photo stored in the input S3 bucket, we trigger LF1 to predict a hair segmentation mask using our SageMaker endpoint with this photo, which has a True value for pixels predicted as hair and False elsewhere.

To dye hairs in different colors, we define an RGB color list that contains many popular hair colors for dyeing [3], such as light pink, light blue, etc. Then, we randomly select three RGB colors from this list. For each randomly chosen RGB color: we first use the predicted hair mask and this color to create a colored hair mask, which has corresponding RGB values in pixels predicted as hair; then, we combine this colored hair mask and the original user's photo with well-determined coefficients to generate a dyed hair photo. After this step, we have three generated dyed hair photos with different random hair colors. We store these photos in the output S3 bucket, which updates all previously generated photos with our newly generated ones. In this way, we make sure the output S3 bucket only keeps three generated dyed hair photos for the user's latest uploaded photo.

4.3.2 Return Recommendations Lambda Function (LF2). LF2 accesses the output S3 bucket and gets the URLs of all three dyed hair photos in the bucket generated by LF1 with the user's latest uploaded photo. After that, we return a list of these dyed hair photo URLs in JSON format in the response body to the API Gateway.

4.3.3 Search Hair Salons Lambda Function (LF3). LF3 searches the nearest barber shops by the zip code. Using the input zip code from the API Gateway, we first search all barber shops whose zip code falls within the range between *max(Manhattan smallest zip code of 10000, input zip code − 20)* and *min(Manhattan largest zip code of 11697, input zip code + 20)* in OpenSearch. For example, if the input zip code is 10025, we will search barber shops with zip codes in [10005, 10045]. After we get barber shop business ID(s) with zip codes returned from the OpneSearch result, for each barber shop, we also get its business name, address, phone, rating, and an image

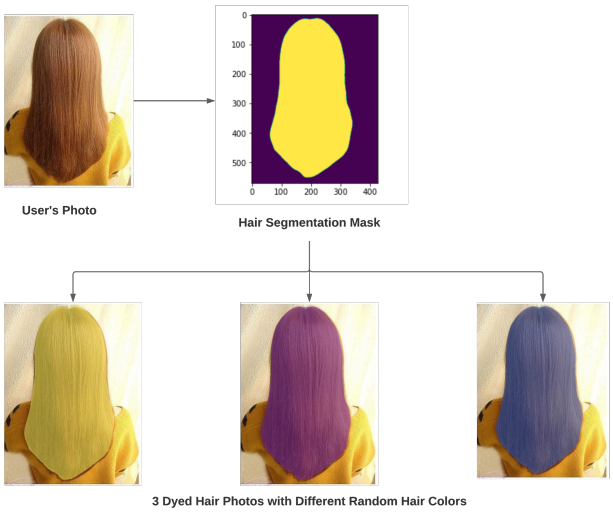

Figure 4: LF1 Flow of Dyed Hair Photos Generation

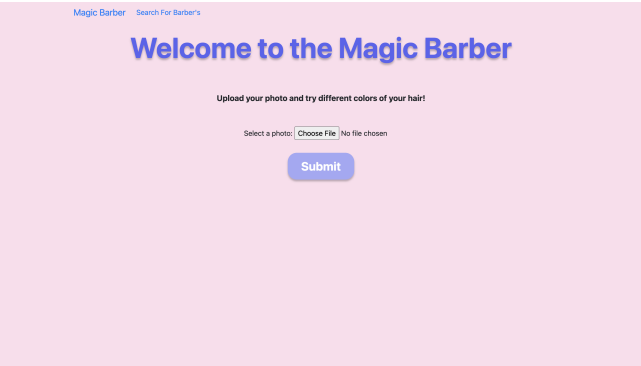

Figure 5: Home Page

URL from DynamoDB using the business ID as the key, to form a final list with complete nearby barber shops data. Then, we sort the final list we make with the absolute difference between each barber shop's zip code and the input zip code, in order to display the resulted barber shops from the nearest to the farthest in the distance in the frontend search page to the user. In the end, we return the final list in JSON format in the response body to the API Gateway.

4.4 Frontend

Magic Barber is broken down into three web pages, and each of them solves one user goal. The home page (Figure 5) receives a hair photo from the user, and it is connected with the `PUT/upload` method. The recommendation page (Figure 6) gives back the generated dyed hair photos to the user, and it is connected with the `GET/results` method. The search page (Figure 7) gets query info and sends back the info that the user needs, and it is connected with

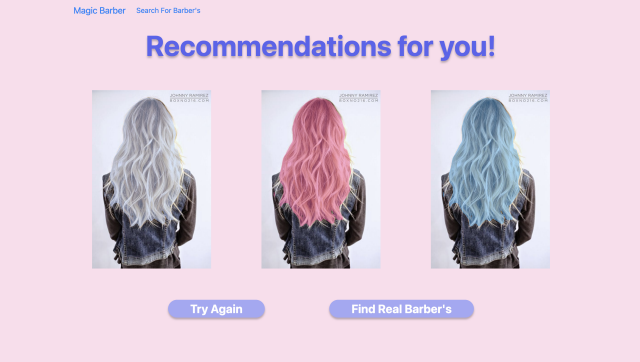

Figure 6: Recommendation Page

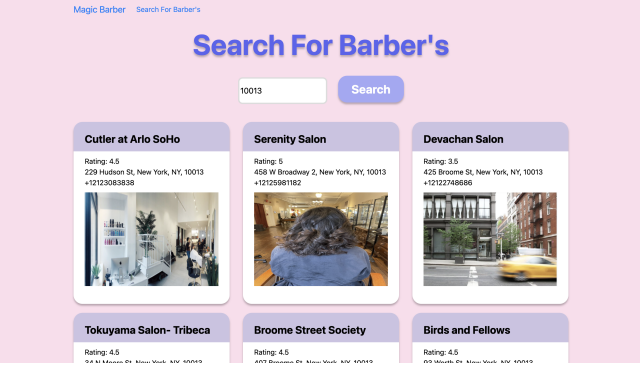

Figure 7: Search Page

the `GET/search` method. Hence, each frontend page is connected with one API method to complete the task.

5 CONCLUSION

Integrating a series of AWS services like Simple Storage Service (S3), API Gateway, Lambda, SageMaker, OpenSearch, and DynamoDB, Magic Barber successfully solves the problem of what color to dye and where to do hair dyeing with efficient request handling, fast data transfer, perfect model inference, scalable data storage, and distributed searching. The exquisite design of architecture, taking full advantage of the cloud services, ensures the stable operation of the web application.

Following from the current Magic Barber, there are a few potential advanced features that can be considered to enlarge our project's scope:

Extending the choice of hair colors: Allowing users to have a wider range of colors to choose can be implemented in our project. Instead of recommending three random hair colors to the user, visualizing a list of colors on the recommendation page and allow the user to choose some colors and try them on virtually would be helpful.

Social media connections: Connecting Magic Barber with social media platforms can also be implemented in the future. This feature would allow users to share the recommended hair colors to their friends through social media platforms such as WhatsApp, Instagram, Facebook, etc.

REFERENCES

[1] *Get started with the yelp fusion API.* URL: https://www.yelp.com/developers/documentation/v3.

[2] Joong Kim. *Ybigta/Pytorch-hair-segmentation: Pytorch-hair-segmentation.* Aug. 2020. URL: https://github.com/YBIGTA/pytorch-hair-segmentation.

[3] *RGB color Hex and Decimal codes table.* URL: https://flaviocopes.com/rgb-color-codes/.

[4] Michele Svanera et al. "Figaro, hair detection and segmentation in the wild". In: *2016 IEEE International Conference on Image Processing (ICIP).* IEEE. 2016, pp. 933–937.

[5] Hengshuang Zhao et al. "Pyramid scene parsing network". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2017, pp. 2881–2890.