Eat-out Assistant

A web application for restaurant recommendation, order, and review

Haoran Guo
Department of Electrical Engineering
Columbia University
hg2461@columbia.edu

Minhui Li
Department of Electrical Engineering
Columbia University
ml4026@columbia.edu

Junyi Chen
Department of Electrical Engineering
Columbia University
jc4805@columbia.edu

Yamei Zhu
Department of Electrical Engineering
Columbia University
yz3167@columbia.edu

***Abstract*—This Project implement a web application for restaurant recommendation, order and review based on various Amazon Web Cloud: S3, API Gateway, Lambda, SQS, SNS, CloudWatch, Elasticsearch Service, DynamoDB, Lex, Comprehend and Rekognition. Based on the service, users can order dishes, make reviews with text, rating and upload selfies, and get recommendation with our services.**

Keywords - Cloud Computing; Amazon Web Service; Restaurant Recommendation; Ordering; Review

I. Introduction

Nowadays, with the development of Internet web service and related technologies, food service application has been popular. And due to people's time limitation and transformation of modern life style, people prefer ordering online rather than going outside or do the cooking by themselves.

Cloud Computing is an Information Technology which enables ubiquitous access to shared pools of system resources and high level of services with minimal management efforts. With Cloud Computing, we can easily develop a web service without the configuration of server and realize lots of powerful functions with the service provided by Cloud Platform.

There are many Cloud Computing services provided by different companies, like Amazon Web Service, Microsoft Azure, Google Cloud, Alibaba Cloud, etc,. Each Cloud Service Platform have its own advantages, and we apply Amazon Web Services to our project due to its broad class of services. In our project, we aggregate various services provided by Amazon AWS: S3, API Gateway, Lambda, SQS, SNS, CloudWatch, Elasticsearch Service, DynamoDB, Lex, Comprehend and Rekognition.

In this project, we decide to develop a web application which make use of the advantages of Cloud Computing and Amazon Web Service and integrate the advantages of popular food service application like Yelp, Meituan and make it more people oriented.

With our "Eat-out Assistant", users can make order requests with certain dishes in a restaurant, and order text message will be sent to the staff of corresponding restaurants. Also, users can make reviews to restaurants in three ways: Rating, text review, and upload selfies. Our backend will compute a restaurant's rating with those three elements and make corresponding updates in real time. Besides, our web services will recommend restaurants of certain type or cuisine according to the rating of the restaurant.

II. System Overview

Figure 1 below shows the whole architecture of our web application realization.

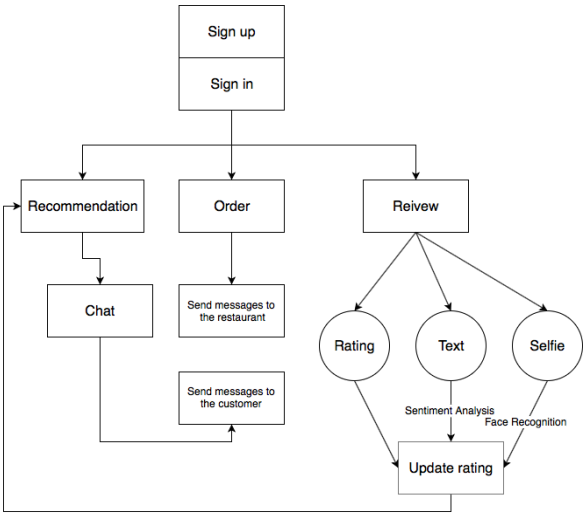

Figure 1. Architecture Overview

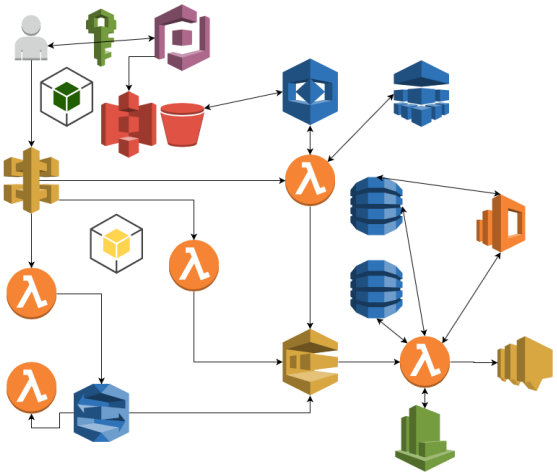

Figure 2. Service Flow of the whole system

Figure 2 shows the AWS services applied to the application and their interaction in each module.
First, users need to signup and sign in through AWS Cognito to have the user authentication to access our service.
Our application consists of three modules: Recommendation, Order and Review. Each of the module is completed by a Lambda function, users make requests to corresponding lambda through API Gateway when interacting with the frontend.

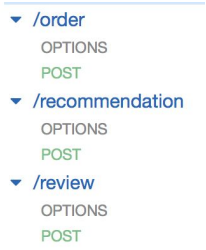

Figure 3. API Gateway

For Recommendation, users can interact with the Lex Bot to offer the preference for restaurants. Our service will poll the messages from SQS, query the ElasticSearch for the top recommended restaurants ID and then search the DynamoDB for related restaurants details, and then composed text messages to send to users.
For Order, users make orders to related restaurants for certain dishes, then the backend searches DynamoDB for the prices of ordered dishes and calculates the total price and sends messages to the staff of corresponding restaurant's staff.
For Review, users can make reviews to a restaurant in three ways: star rating, text review and upload selfies. We make use of AWS Comprehend to analysis text review, and AWS Rekognition for image analysis. We will compute the total score with those three kinds of ratings and apply it to the recommendation function.
More important details of each module are described in the following chapters.

III. REVIEW

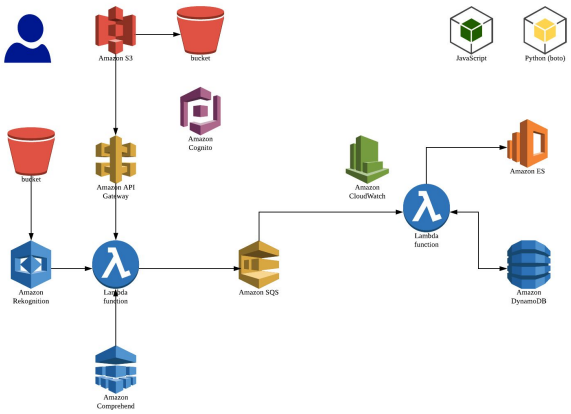

Figure 4. Review Module Architecture

In the review module, users can make reviews to a certain restaurant in three ways; rating, text review and upload selfies in the restaurant.

A.Rating

For rating, users simply give a score to the restaurant in the range of 1 to 5, which is direct and obvious.

B.Text sentiment analysis - AWS Comprehend

To process the text review users submit, we use Amazon Comprehend APIs. Amazon Comprehend is a Natural Language Processing service that uses combined Machine Learning techniques to find insights and relationships in text. It has the ability to identify the language of the text; extract key phrases, places, people, brands, or events; understand how positive or negative the text is; and automatically organize a collection of text files by topic. We use Amazon Comprehend APIs to analyze the sentiment of the texts from users to judge how positive or negative the texts suggest. For a given sentence or several sentences, the API can give us the confidence for each of the four sentiments, which are positive, negative, mixed and neutral. We simply subtract the confidence of positive by that of negative to get a score and multiply it by a weight as the value change to the original rate.

```
#Comprehend
try:
    res = comprehend.detect_sentiment(Text = text, LanguageCode = 'en')
    print("Comprehend succedded")
    print("Comprehend", res)
    if 'SentimentScore' in res:
        sentscore = res['SentimentScore']
        positive = sentscore['Positive']
        negative = sentscore['Negative']
        hidrate += 0.25 * (positive - negative)
except:
    print("Comprehend failed")
```

Figure 5. Code snippet for text sentiment analysis

C. Image analysis - S3 & AWS Rekognition

Users upload their photos from frontend to an S3 bucket. Before uploading process, user should get permission to write on S3, this step could be authenticated through Cognito. In the Lambda function, we attached the corresponding policies to allow Lambda to retrieve the photo from buckets, using bucket name and file key.

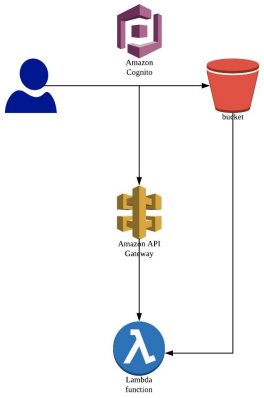

Figure 6. Flow of uploading photos to S3 bucket

To process the uploaded selfie, we use Amazon Rekognition APIs. Amazon Rekognition makes it easy to analyze images in the S3 bucket in this application. The service can identify objects, people, text, scenes, and activities, as well as detect any inappropriate content. Amazon Rekognition also offers highly accurate facial analysis and facial recognition. It is based on the same proven, highly scalable, Deep Learning technology.

Assume that the uploaded selfie has a face on that. Amazon Rekognition can judge if the face is smiling and give the confidence. It returns the analysis of mood as well. The moods contain happy, sad, angry, confused, disgusted, surprised, calm and unknown, each with a confidence. We select happy, sad, angry and disgusted, which are useful for rating the restaurants, and multiply the confidence with a self-defined score (shown in Table 1) as part of the change value of the original rate.

Emotion	HAPPY	SAD	ANGRY	DISGUSTED
Score	1	-0.5	-0.9	-1

Table 1. Self-defined score for image rating

In total, combine the text analysis and facial analysis together with the original integer star rating, we finally update the hidden rating in the database according to the equation below. Our recommendation is based on this hidden rating in the database.

$$\begin{aligned} Hidden\_rate = Star\_rates \\ + 0.25(positive - negative) \\ + 0.1 \times (smile - notsmile) \\ + 0.15 \times emotion\_score \times confidence \end{aligned}$$

Figure 7. Formula for hidden_rate

The following code snippet shows the image analysis part.

```
#Rekonition
try:
    #print("bucket", BUCKET_NAME)
    #print("key", KEY)
    response = Rekclient.detect_faces(Image = {'S3Object':{'Bucket': BUCKET_NAME, 'Name': KEY}}, Attributes = ['ALL'])

    print("Reknization succeeded")
    print("Reknization", response)

    if 'Smile' in response:
        if response['Smile']['Value']:
            hidrate += response['Smile']['Confidence']

    if 'Emotions' in response:
        for emotion in response['Emotions']:
            e_type = emotion['Type']
            conf = emotion['Confidence']
            if e_type in score_dict:
                hidrate += score_dict[e_type] * conf
except ClientError as e:
    print("Reknization failed")
    print(e)
```

Figure 8. Code snippet for text sentiment analysis

Rate Update

After computing the “hid_rate” with weight of those three different kinds of score, the information will be sent to SQS with label “rate”.

The core lambda function “LF2” which is triggered by CloudWatch poll message from SQS every minutes. When detecting the “rate” label, it will parse the information “hid_rate” and update it in DynamoDB and Elasticsearch. The following code snippet shows how we update the rate of restaurants of the table “my_restuarant” (which architecture shows in Figure 11.) in DynamoDB and Elasticsearch index “elastic_restuarant” (which architecture shows in Figure 12.).

The new rate updated in the Elasticsearch service will impact the recommendation result for the next time.

```
# Update Dynamo
try:
    response = table.update_item(
        Key={
            "ID": id

        },
        UpdateExpression='SET rate = :val1, hid_rate = :val2, reviewnum = :val3',
        ExpressionAttributeValues={
            ':val1': update_rate,
            ':val2': update_hid_rate,
            ':val3': old_reviewnum+1
        })
except ClientError as e:
    print(e.response['Error']['Message'])
else:
    print("Update succeeded:")
    print("Updated", item)
```

Figure 9. Code snippet for updating DynamoDB

```
#Update Elastic Search
try:
    es.update(index='elastic_restaurant',doc_type='search',id=id,
            body={
                "doc":{
                    "hid_rate": new_hid_rate,
                    "rate": new_hid_rate
                      }})
except:
    print("Updating ES failed")
else:
    print("ES updating succeeded")
```

Figure 10. Code snippet for updating Elasticsearch

ID	cuisine	dish1	dish2	dish3	hid_rate	name	phone	rate	reviewnum	zipcode	address
2	Chinese	4	5	6	3.44110...	Szechuan Garden	5187728693	3.44	5	10025	
8	Americ...	22	23	24	3.8	Shake Shack	5187728693	3.8	8	10025	
9	Americ...	25	26	27	4.2	Five Guys	5187728693	4.2	9	10025	
1	Chinese	1	2	3	4.3	Happy Hot Hunan	5187728693	4.3	1	10025	
6	Indian	16	17	18	3.8	Masala Club	5187728693	3.8	6	10025	
5	Indian	13	14	15	3.74697...	Roti Roll Bomba...	9179124385	3.66666...	9	10025	
4	Chinese	10	11	12	4.6	Xi'an Famous Fo...	5187728693	4.6	5	10025	
7	Indian	19	20	21	4.1	Aangan	5187728693	4.1	7	10025	

Figure 11. Architecture of "my_restuarant" table

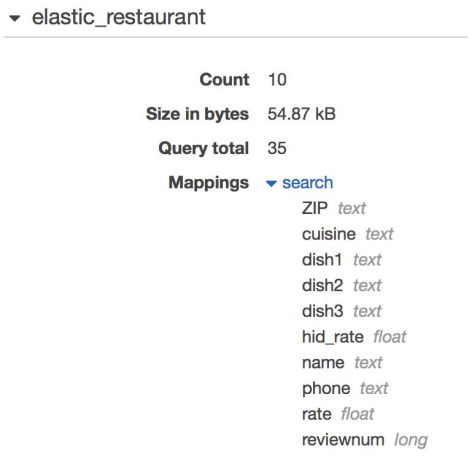

Figure 12. "elastic_restaurant" index and mapping.

Result

A. Making positive review

We make a positive review on the "Xi'an Famous Foods" restaurant:

1. Rating: 5

Figure 13. Positive rating

2. Selfie with happy face:

Figure 14. Positive selfie

3. Positive text review:

The food is great! I love it !

Figure 15. Positive text review

Result:

Xi'an Famous...	5187728693	4.6	5

Figure 16. Original record for "Xi'an Famous Food"

Xi'an Famous...	5187728693	4.66666666667	6

Figure 17. Updated record for "Xi'an Famous Food"

We show part of content in the table and the third column is rate and the fourth column is review number. We can see that the rate for the "Xi'an Famous Foods" is updated from 4.6 to 4.66666667 and the review number changes from 5 to 6.

B. Making negative review

We make a negative review on the "Happy Hot Hunan" restaurant:

1. Rating: 2

Figure 18. Negative rating

2. Selfie with sad face:

Figure 19. Negative selfie

3. Negative text review:

The food is terrible, I won't go here again...

Figure 20. Negative text review

Result:

Happy Hot H...	5187728693	4.325	4

Figure 21. Original record for "Happy Hot Hunan"

Happy Hot H...	5187728693	4.26	5

Figure 22. Updated record for "Happy Hot Hunan"

IV. ORDER

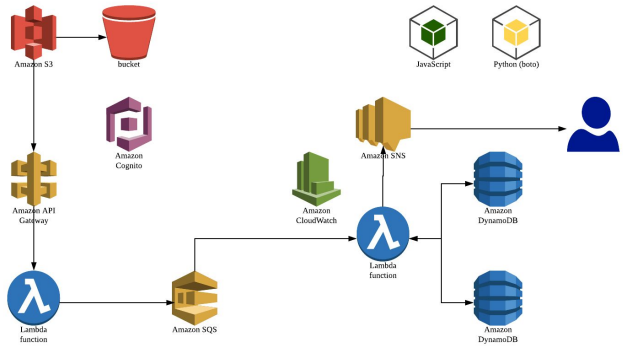

Figure 23. Order Module Architecture

In this part, users can choose 2 dishes to order from certain restaurant. Corresponding restaurant's id, dishes' ids and amounts will be posted through API Gateway to "Order_lambda" function. The function will pack the information and send it to SQS with label "order".

The core function "LF2" poll messages from SQS per minutes, when recognizing the label "order", it will parse the restaurants' ID, dishes' ID and related amount from the message.

First we query "my_restaurant" table for restaurant name and contact phone number with restaurant's ID.

```
# Querying From Dynamo_table_restaurant
try:
    response = table.get_item(
        Key={
            'ID': str(order_id)
        })
except ClientError as e:
    print("Dynamo query error!")
    print(e.response['Error']['Message'])
    pass
else:
    item = response['Item']
    print("GetItem succeeded:")
    item_json = eval(json.dumps(item, indent=4, cls=DecimalEncoder))
    print("item_json", item_json, type(item_json))
```

Figure 24. Code snippet for querying "my_retuarant" table

Then we query "my_dishes " table for dishes' name and price. Figure 26. Shows and architecture of "my_dishes" table.

```
# Querying From Dynamo_table_dishes
def query_dish_db(table_name, dish_id):
    table = dynamodb.Table('mydishes')
    try:
        print("dish_id", dish_id)
        response = table.get_item(
            Key={
                'ID': str(dish_id)
                })
    except ClientError as e:
        print("Dynamo_table_dishes query error!")
        print(e.response['Error']['Message'])
        pass
    else:
        item_dish = response['Item']
        item_json_dish = eval(json.dumps(item_dish, indent=4, cls=DecimalEncoder))
        return item_json_dish
dish1_json = query_dish_db('mydishes',item_json["dish1"])
dish2_json = query_dish_db('mydishes',item_json["dish2"])
if amount1:
    message += dish1_json['name'] + ", amount:" + str(amount1) + ".\n"
if amount2:
    message += dish2_json['name'] + ", amount:" + str(amount2) + ".\n"
```

Figure 25. Code snippet for querying "my_dishes" table

	ID	name	price
	22	ShackBurger	5.69
	18	Ragda Patties	7
	16	Masala Dosa	9
	2	Orange Beef	16.9
	13	Chicken Lolip...	6
	8	Chicken and ...	7.81

Figure 26. Architecture of "my_dishes" table

Finally, we compute the total price of order, and send a text message through SNS to the restaurant staff, then the order making process is completed.

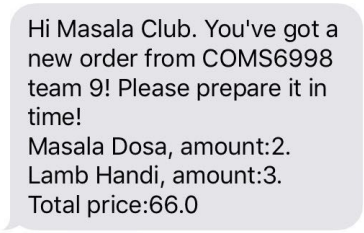

Figure 27. Text message received by restaurant staff

V. RECOMMENDATION

This part is inherited from our Assignment 2 & 3, so we will not include too much details for this part.

Users can interact with the Lex Bot to offer preferences for restaurants: cuisine, location, time, number of people and phone number. We trained the Lex Bot to collect those messages and send to the SQS message queue.

Our core Lambda function (LF2) poll the message labelled "recommendation" from SQS and parse the important message. We use "cuisine" to query the Elasticsearch service for the top 2 restaurants' id with highest "hid_rate". The Elasticsearch index and the mappings are shown in the Figure 12.

```
#search es
res = es.search(index="elastic_restaurant", doc_type="search", body=
    {
        "from": 0, "size":2,
        "query":
            {
                "match":
              {
                 "cuisine": term
              }
            },
        "sort":[
            {"hid_rate":{"order":"desc"}}
        ]
    }
    )
result_id=[]
print("%d documents found" % res['hits']['total'])
for doc in res['hits']['hits']:
    print("%s %s" % (doc['_id'], doc['_source']['cuisine']))
    result_id.append(doc['_id'])
```

Figure 28. Code snippet for searching the Elasticsearch

After getting the restaurant's ID, we can query the "my_restuarant" table for details.

After having the details of desired restaurants, we send text message through SNS services.

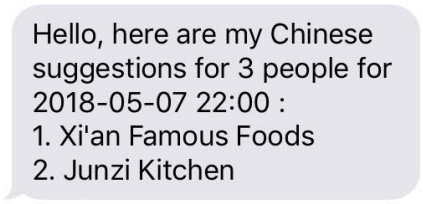

Figure 29. Text message received by users showing the recommendation result

The Recommendation module is composed in "LF2" lambda function which is triggered by CloudWatch service for every minutes.

VI. FRONTEND

Our frontend web page is composed with JavaScript, it is nice-looking and practicable. Each module are shown on the webpage for easily use.

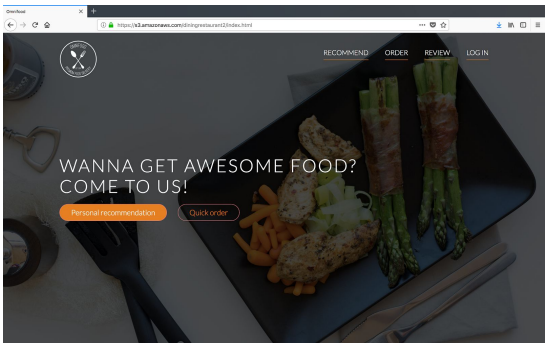

Figure 30. Welcoming page

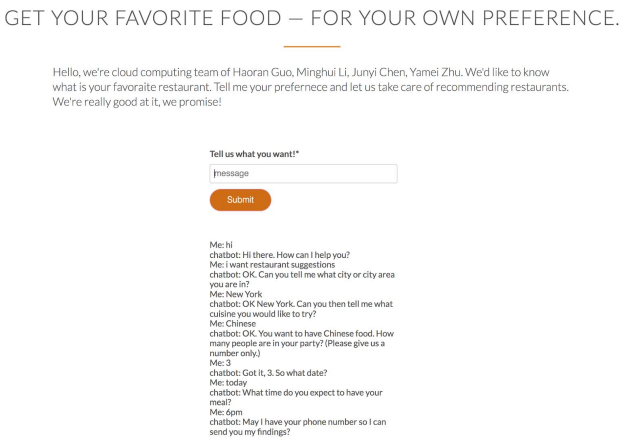

Figure 31. Recommendation Module

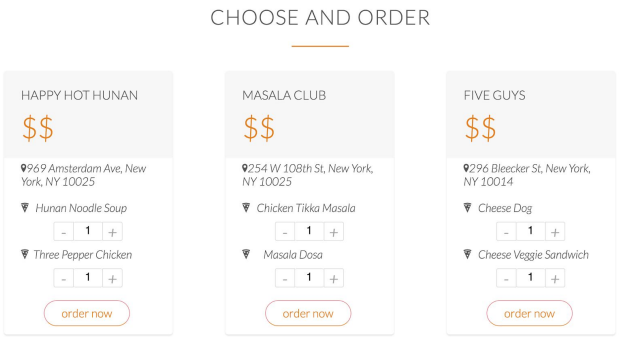

Figure 32. Order Module

Figure 33. Review Module - Rating

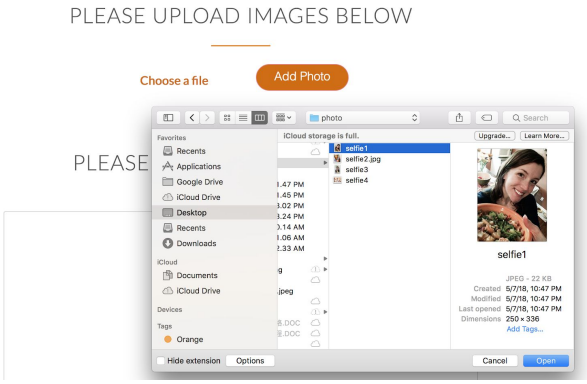

Figure 34. Review Module - Photo Review

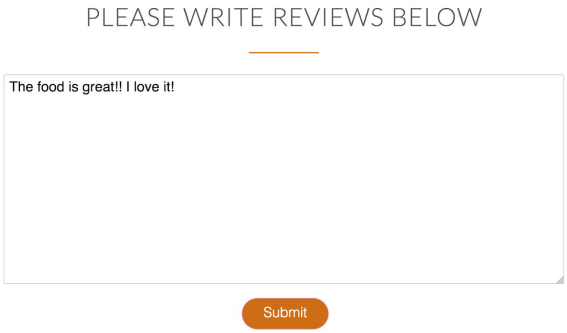

Figure 35.. Review Module - Text Review

VII. Conclusion

In our project, we implement a food service application with three basic modules: Order, Review and Recommendation. Users can make order for a certain restaurant with different dishes and restaurant staff will receive the order.

Users can make review with rating, upload selfies with different expression with their feeling about the restaurant and write text reviews. Our backend will update the new rate of restaurant with the computation of those three kinds of reviews with different weight. The DynamoDB and Elasticsearch will be updated in real time for recommendation.

For recommendation, users get recommendation with their preference with the two restaurant that have highest scores.

We think this application is useful and practical, but there are still many improvements we can make.

For future work, we would like to set up a personal center for users to see their historical records, which might require establishing separate table for each different user.

With users' historical data, we can do collaborative filtering or more advanced recommendation algorithm, then we can analyze similar users to recommend similar restaurants.

In the order part, our future ideas include building up an searching engine to query for restaurant and dishes details from the front end. Users could search for corresponding information about any restaurant in our database. Also, setting up an interaction between users and restaurant owners could support other functions, such as take-out service and online payment services

Last but not least, we are really grateful for this course, for that we have learnt many things about Cloud Computing and big data analytics and we have done a lot of work to enhance our background in this area.

References

[1] https://en.wikipedia.org/wiki/Cloud_computing. 2018.

[2] Amazon Comprehend Developer Guide.. https://docs.aws.amazon.com/comprehend/latest/dg/what-is.html. 2018.

[3] Amazon Rekognition Developer Guide. https://docs.aws.amazon.com/rekognition/latest/dg/what-is.html. 2018

[4] Amazon Elasticsearch service Developer Guide. https://docs.aws.amazon.com/elasticsearch-service/latest/developerguide/what-is-amazon-elasticsearch-service.html. 2018.

[5] Radu Gheorghe, Matthew Lee Hinman, Roy Russo. Elasticsearch in Action. 2016.

Link to Github: https://github.com/yamei0811/Restaurant-Order-Assistant

Youtube Link: https://youtu.be/5oHJ4hFRqt4