

**UPMC HEALTHY DIET EVALUATION SYSTEM**

WITH AMAZON WEB SERVICE

Team 14:

Yujie Qiu

Ji Wang

Mingquan Zhang

Xian Zhang

Zhiyuan Zhang

**Table of Contents**

[Executive Summary 4](#_Toc494232217)

[Design of the System 5](#_Toc494232218)

[**Image Retrieval from Users** 5](#_Toc494232219)

[*Overview of the Implementation* 5](#_Toc494232220)

[*Implementation Details* 5](#_Toc494232221)

[*Performance Testing* 6](#_Toc494232222)

[**Image Recognition and Evaluation** 6](#_Toc494232223)

[*Overview of the Implementation* 6](#_Toc494232224)

[*Implementation Details* 7](#_Toc494232225)

[*Accuracy Evaluation* 9](#_Toc494232226)

[*Performance Testing* 11](#_Toc494232227)

[**Display of Users’ Diet evaluation** 12](#_Toc494232228)

[*Overview of the Implementation* 12](#_Toc494232229)

[*Implementation Details* 13](#_Toc494232230)

[*Performance Testing* 13](#_Toc494232231)

[Cost Analysis and Projections 13](#_Toc494232232)

[*One User* 14](#_Toc494232233)

[*Increments of 1000 Users* 15](#_Toc494232234)

[*Increments of 1 Million Users* 16](#_Toc494232235)

[Limitations of the System 18](#_Toc494232236)

[Scale-up Possibility 19](#_Toc494232237)

[Appendix 1 20](#_Toc494232238)

[Appendix 2 21](#_Toc494232239)

[Appendix 3 22](#_Toc494232240)

[Appendix 4 23](#_Toc494232241)

[Appendix 5 24](#_Toc494232242)

[Appendix 6 25](#_Toc494232243)

[Appendix 7 26](#_Toc494232244)

[Appendix 8 27](#_Toc494232245)

[Appendix 9 28](#_Toc494232246)

[Appendix 10 29](#_Toc494232247)

[Appendix 11 30](#_Toc494232248)

[Appendix 12 31](#_Toc494232249)

[Appendix 13 32](#_Toc494232250)

[Appendix 14: Design Document 33](#_Toc494232251)

[a) A detailed description of the system and needed functionality 33](#_Toc494232252)

[b) A detailed list of the system requirements 34](#_Toc494232253)

[c) Assumptions made for the simulated system: 34](#_Toc494232254)

[d) An architectural design of the proposed system 35](#_Toc494232255)

[e) Tools, technologies and cloud services needed & evaluation 36](#_Toc494232256)

[f) Realistic scenarios of end-to-end test 42](#_Toc494232257)

[References 43](#_Toc494232258)

# Executive Summary

UPMC is now exploring possibilities to evaluate clients’ healthy lifestyle and use this information to offer health insurance discounts to clients who lead healthy lifestyles. In this report, we demonstrate our system design to evaluate UPMC clients’ healthy lifestyle mainly based on their diet choices.

In our design, the sensing device that users wear will automatically capture and upload the food images of the users into the AWS S3 bucket for storage. After that, users submit their user IDs to the website that we designed. On the webpage, users and UPMC can easily attain a diet evaluation report including pie chart and point chart to grade and analyze the dietary habit of users. We achieve this functionality by deploying systems on AWS (Amazon Web Services), the detailed process is as follows. ***Appendix 1*** also shows a flowchart for our architecture design.

Firstly, all food images will be uploaded to the S3 bucket on AWS. Since we use Lambda function to run the code and evaluate users’ food images, SNS (Simple Notification Service) is a great bridge between S3 bucket and Lambda. When the S3 bucket sends a request to SNS, SNS will notify and send an event to Lambda. After lambda gets the notification, it will execute the code by calling Amazon Rekognition API to recognize photos uploaded to the S3 bucket. After this, all food images will be labeled with different tags. Then, we design an algorithm to assign different scores to each tag and calculate a health score for each food image. Those scores will then be stored to DynamoDB (database). When users or UPMC want to check their dietary report, they are able to log in the website that we designed. After submitting their user IDs through the website, the server on the EC2 (Elastic Compute Cloud) will post their user ID to the DynamoDB. On the DynamoDB, user ID will be matched out and push the user’s daily meal score back to the server for exhibition on the website.

# Design of the System

## **Image Retrieval from Users**

### *Overview of the Implementation*

The realistic process for this part is like the following: a client is taking her meal; a camera will capture an image of the food that she is eating. Then this image will be uploaded to the S3 Bucket. In our simulated system, we directly start from uploading images to the S3 bucket. The process of uploading images from cameras to S3 Buckets is not included in our simulation system. When S3 Bucket detects that a new object appears in it, it will notify SNS and ask to publish this event. SNS will then publish this event to the designated Lambda function. And then this Lambda function will grab the newly uploaded image into its space for later use.

### *Implementation Details*

The image of food can be captured either manually by a nurse or automatically by devices. S3 Bucket is a web service that can be used for storage. The S3 Bucket should be configured to be able to detect a newly appeared item. This is done in the Bucket policy, where the bucket configuration is shown in ***Appendix 2***. This configuration means that the configuration itself is not restricted to a typical type, and it will monitor if there is a new object of any type created in it. In this way, the S3 Bucket itself will actively see if there is a new object. Moreover, the bucket should be added with an Event, which will send the information about this newly created object to SNS.

SNS is the service that monitors the publication of events. It should be added to a topic corresponding to the Event added to S3 Bucket so that it will grant the bucket with the permission to publish events through it.

For the Lambda function, it should be configured that it can be triggered by SNS service, and also the very sns topic. In this way, when a new object is detected in the S3 Bucket, the Lambda function will be triggered. Meanwhile, sns will send it a “event” parameter that contains the information about this trigger. The ‘event’ parameter is a json dictionary. Information that the Lambda function will need is stored in a Json string with the key named “message”. Lambda can read the bucket name and newly appeared item name through this “message”. Therefore, Lambda can download the newly uploaded item into its space for later use.

### *Performance Testing*

We apply one unit test in this process. In the Lambda function, we log the “event” parameter into the system log. And then we upload one image into the AWS S3 bucket to trigger the Lambda function. We check the logs on the CloudWatch to see if the Lambda function has been triggered (e.g. the input and output of the Lambda function, any error message occurs). The unit test passed as during the test, the logged event parameter is in the correct structure and the name of bucket and image are documented.

## **Image Recognition and Evaluation**

In this section, we will describe how our simulated system uses Lambda functions to call Amazon Rekognition API to analyze images and evaluate users’ healthy diet conditions.

### *Overview of the Implementation*

In general, our simulated system works as follows. After the Lambda function receives the event from SNS, it will then grab the food images from the S3 Bucket and call the Amazon Rekognition API to evaluate the food images. The Amazon Rekognition API will return tags/labels for each food image including a confidence level indicating how likely the food image will fall into each tag/label. A demonstration of this API is displayed in ***Appendix 3***. Then we design an algorithm to analyze the tags and confidence level to return a healthy diet evaluation score for each food image. After the evaluation score is calculated for each image, we then store these scores in Dynamo database for further process.

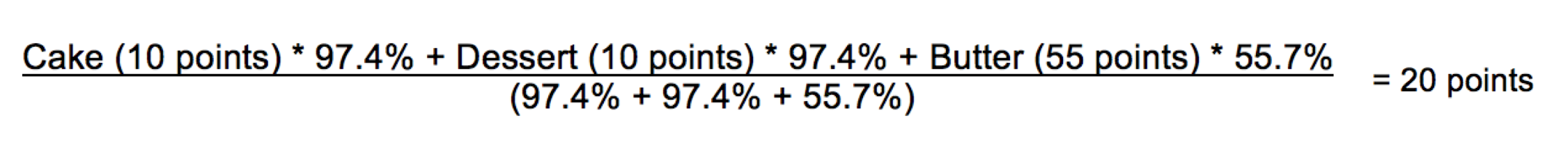
### *Implementation Details*

For the detailed implementation inside the Lambda function, one input parameter for the Lambda Function is “event”, which is the very information the SNS sends to it. “Event” is a Json dictionary, which contains all the information about the event. For this system, the information that the Lambda Function needs are all stored in the key named “message”. “Message” is a Json string. When “message” is loaded, Lambda Function can obtain the name of the newly uploaded image. It will then grab the image from the S3 Bucket and call the “detect\_labels” function of the Amazon Rekognition API. The “detect\_labels” function will return labels for the uploaded food images. Each label will mainly include two fields: the name of the label and the level of confidence that the image contains the object. When using the “detect\_labels” function, we included a minimum confidence level filter to only return the labels with confidence level higher than or equal to 50%. The reason we choose 50% is that during our test of evaluating more than 200 images, all effective labels are above 50% confidence level. Also in Amazon Rekognition Developer Guide, the default suggested minimum confidence level is 50%.

After we get the name and confidence level field of the image labels, we then implement an algorithm to evaluate and score the food images based on their labels. To begin with, we scan through all labels return by the images to check if the ‘Food’, ‘Vegetable’, ‘Fruit’ or ‘Drink’ label is included. If no such labels are included, this means the images uploaded are not food and we will return “-1” for such images. Then, we filter all label names and exclude the labels which are irrelevant to food, such as dish/plate/bowl and other labels that are too general and not deterministic, such as food. The details of our algorithm work as following. We assume users’ healthy diet scores have a total of 100 points, with 0-40 as unhealthy, 40-70 as in fair health conditions and over 70 points as healthy. Then, we categorize all label names into three types: the first type is healthy and we assign 90 points for all labels in the healthy category; the second type is unhealthy and we assign 10 points for all labels in the unhealthy category; the rest are foods that are neither healthy nor unhealthy and we assign 55 points for all labels in this category. To determine which food items are healthy, we mainly include all fruits and vegetable. We list all healthy fruits and vegetable labels in a set and check whether each label returned fall into the healthy set. If the label falls into the set, we assign 90 points to that label. To decide which foods are unhealthy, we mainly consider fried foods, desserts and junk foods. Also, we list all unhealthy foods in a set and check whether each label returned fall into the unhealthy set. If the label falls into the set, we assign 10 points to that label. For the rest of the labels that neither falls into the healthy set or the unhealthy set, we assign 55 points to them.

For instance, to analyze the labels returned by the image shown in ***Appendix 4****,* we use the following approach. The “Food” label is too general and thus is excluded from our consideration. Now, there are three labels left: “Cake”, “Dessert” and “Butter”. Among the three labels, “Cake” and “Dessert” belong to the unhealthy category and therefore, we assign 10 points for these two labels. While the “Butter” label belongs to the third category that is neither healthy nor unhealthy, therefore we assign 55 points to “Butter”.

After assigning scores for each label, we now focus on how to combine the scores for each label in order to get a final healthy diet evaluation of the food image for each user. Our algorithm is as following: each label has its own confidence level and we get the total score for each image by calculating the weighted average of the label scores based on the labels’ confidence level. The following example will illustrate this process.

Again, take ***Appendix 4***for example, the calculation is as follows:

From this illustration, the healthy diet evaluation score for the cheesecake is 20 points. Since 20 points fall between 0 – 40 points, we draw the conclusion that cheesecake is an unhealthy diet.

After we evaluated the healthy diet scores for each food images, we then store the scores for each image into Dynamo Database. We realize this process by calling the ‘put\_item’ function on the Dynamo Database and then write the score data into the score table in Dynamo Database. In the Dynamo Database, we store and manage two tables. The first table is a user ID table, where we match each user’s name with their user ID. The user names are specified when users upload images into the S3 buckets. The second table is a score table for each user, where we match our healthy diet scores with each image at a certain time point. Dynamo Database stores this information and then interacts with the EC2 server in order to send these information for display on the server’s URL. The detailed description of this process is discussed in the next section.

### *Accuracy Evaluation*

In this part, we will evaluate the accuracy of our healthy diet scoring system. There are three parts of our accuracy evaluation. First, we will prove why weighted average calculation based on confidence level is more accurate than simple average. Then, we will discuss the logics for setting up the points for healthy and unhealthy foods and explain why such points allocation will lead to an accurate diet evaluation system. Last but not least, we will show the accuracy of our healthy diet evaluation system when dealing with multiple food items in one image.

To begin with, our system calculates the point for food images by taking the weighted average score of each food labels’ points. This algorithm can provide more accurate results since we assign higher weights to labels that better describe the foods in the images. Whereas, taking simple average will assign the same weight to all labels and thus the points of some less accurate labels will affect the overall accuracy of our algorithm. Here is a detailed example to illustrate the accuracy of our weighted average approach. The main labels we take into account here are “Burger” and “Salad”, other labels are either irrelevant or too general to be considered. In ***Appendix 5***, if we use simple average method, we will get a total of (Burger (55 points) + Salad (90 points)) / 2 = 72.5 points. By using the weighted average method, we assign lower weights to “Salad”, which is a not very accurate description of the food images uploaded and therefore can achieve a more accurate evaluation. The calculation is as following: (Burger (55 points) \* 99.3% + Salad (90 points) \* 75.2%) / (99.3% + 75.2%) = 70 points. 70 points is a better description of the healthy condition for the burger and falls into the fair health range in our evaluation system. From this example, we could see that by using a weighted average approach, we could leverage the potential inaccuracy of the Amazon Rekognition API and assign higher weights to more accurate labels, therefore producing more convincing healthy diet evaluation results.

Secondly, we will discuss our logics for assigning scores to healthy and unhealthy foods. Originally, we originally the points for unhealthy food as 30, the points for fair food as 55, and the points for healthy food as 80. When we use these benchmark numbers to calculate the score of the fried chicken (shown in ***Appendix 6***), we got 40.48 points ((97.1% \* 30 + 97.1% \* 30 + 70.1% \* 55 + 70.1% \* 55)/ (97.1% + 97.1% + 70.1% + 70.1%)), which indicates that fried chicken is fair. The major problem with this result is that we overestimate the points for unhealthy foods and underestimate the points for healthy foods. Since in our algorithm model, there are mainly two sets of foods that are either very healthy or very unhealthy, we also need to assign more extreme scores to the food labels. Therefore, we adjusted to assign a lower point of 10 to unhealthy foods and raise the points of healthy food to 90 points. With the adjusted algorithm, we got 28.87 points ((97.1% \* 10 + 97.1% \* 10 + 70.1% \* 55 + 70.1% \* 55)/ (97.1% + 97.1% + 70.1% + 70.1%)), which falls into the unhealthy category and is consistent with our common sense. We tried more than 200 food images with this algorithm, and the results all turned out to be fair and matching our general understand of healthy and unhealthy foods.

Last but not least, our algorithm also proves high accuracy in evaluating food images with mixed food items.  As shown in the ***Appendix 7***, the plate contains more than five food items, and this picture got 46.8 points ((97.1% \* 10 + 97.1% \* 10 + 70.1% \* 90 + 70.1% \* 55)/ (97.1% + 97.1% + 70.1% + 70.1%)), which is consistent with our general understanding. The reason that our algorithm could achieve such accuracy is mainly because we use the weight average approach. Therefore, we can take into account all food labels and calculate a comprehensive healthy diet points. This example shows that our algorithm is able to evaluate different food items in the same image accurately.

### *Performance Testing*

We apply two tests in this process. The first test deals with the file’s name of the pictures uploaded. The second test deals with the successful write of records in the DynamoDB.

During the first test, we find a bug when uploading some pictures into the S3 bucket. An error message will occur when the pictures with white spaces among the filenames are uploaded. After comparing every characteristic, the only difference between successful trial and failed trial is the file name. We fix our code to make sure that we remove the whitespace before uploading.

In the second test, we want to check if the record (photo name, time and score) has been written in the table in the DynamoDB. After uploading a photo in the AWS S3 bucket, we check whether the item is recorded in the table and whether the information is correct (***Appendix 8***).

## **Display of Users’ Diet evaluation**

### *Overview of the Implementation*

We will design a website for users (***Appendix 9***). If a user wants to check their daily dietary habit, he will input the user Id on the website. Once server receives request from user’s website, server will connect to Dynamodb by API, and attain user’s data from Dynamodb. When a user’s data is ready on the Dynamo dB, server will push to the website to exhibit readable information such as the gauge chart and point chart for user. The first gauge chart shows users’ dietary score in the most recent week and the second gauge chart shows users’ dietary score in the most recent month. Users will easily observe their dietary score change in a certain time interval. On the point chart, it exhibits User’s health level records by meal in different time. it is very useful for user to check the dietary score in the different time. if a user wants to know the dietary score in specific time, he/she is able to click and drag in the plot area to zoom in. the health level records by meal in that specific period will show on the web page. Users are able to download the PDF version monthly and annual evaluation report for late deep analysis.

### *Implementation Details*

In our plan, when user type the URL in the browser’s address. Server will a use “GET” request to relocated user to our “index.html” which is web page we designed. User will input his/her login ID on this web page and hit the confirm button. After this user ID will be sent to server on EC2. Server will send “POST” request with user’s ID to Dynamodb by API. The user’s ID will be match out and get the user’s data when Dynamodb receives the information from server. User’s data will reform by highcharts JS to exhibits gauge chart and point chart for user.

### *Performance Testing*

we have been through several times performance test. On the first-time attempt, when we uploaded the same name of pictures to our S3 bucket, there was only one record of picture on the, since the second picture replaced the first picture on the Dynamodb. for fixing this problem, we did some changes on the server. when server attained the same name of picture, we added the sequence number for each same name pictures. Therefore, every picture has their own name. we changed the main key on the lambda from upload time to picture name, every problem has been fixed. Even we uploaded two same name pictures at time, there were two records on the Dynamodb and all pictures would be weighted out and calculated for user.

# Cost Analysis and Projections

We break the cost based on one user, an increment of 1000 users and a projected of 1,000,000 users. In either scenario, the cost consists of the following parts: cost of using AWS S3 bucket, cost of using Rekognition API, cost of using Lambda function, cost of using DynamoDB, and cost of using EC2 instance. The estimation of the cost is based on the assumption that a user will upload 15 images per day, the period we track is one month (31 days). All the prices are based on US-EAST region.

### *One User*

First, we assume there is only one user of our system. We assume we will use Standard Storage for larger storage. The cost for AWS S3 bucket consists of three parts: standard storage, standard PUT/POST request, and standard GET and other requests. The price of Storage is $0.023 per GB, and we assume the average pixel size of a photo is 1MB, and the total number of photos uploaded per month is 465 MB (15 \* 31), so 1 GB per month fits the need. There are a PUT/POST request and GET request associated with each photo, so the total number of requests are 465 each. The unit price of each kind of request is $ $0.005 per 1,000 requests and $0.004 per 10,000 requests accordingly. The total price of AWS S3 bucket per user is **$ 0.03** per month.

Next, we move to the cost of Amazon Rekognition API. The total number of images processed by this API per month is 465, and the unit price is $1.00 per 1000 images, so the total price is **$ 1.00**.

As for the AWS Lambda function, the first 1 million requests per month are free, and the total number of requests is 465, which means we can use AWS Lambda function for free.

As for the cost of Amazon DynamoDB, it consists of three components: Provisioned Throughput (Write), Provisioned Throughput (Read), Indexed Data Storage. The unit price for Provisioned Throughput (Write) is $0.47 per WCU (2.5 million writes per month), and the total number of writes for one user per month is 465 (much smaller than 2.5 million), which leads to the total price for read of $0.47. The unit price for Provisioned Throughput (Read) is $0.09 per RCU (5.2 million reads per month), the total number of read is same as the one for write, so the total price for reads is $0.09. The unit price for Indexed Data Storage is $0.25 per GB, the size of one record in a table is 45 bytes, so the total size would be 41,850 bytes (45 (bytes each record) \* 465 (the number of records) \* 2 (the number of tables)), which is 0.042 MB, so 1GB of storage is far enough. Thus, the total price for DynamoDB is **$ 0.81**.

The type of Amazon EC2 instance we choose to use to on-demand t2.micro, and its unit price is $0.012 per hour. We will keep this instance running throughout the month, so the total hour will be 744 hours (24\*31), so the price is **$8.93**.

Because there is only user, we ignore the cost for maintenance.

Overall, the total price for one user deploying our system is **$10.77**.

### *Increments of 1000 Users*

We now increment our users by 1000. The price for AWS S3 bucket will increase because the storage and the number of requests both increase (***Appendix 10***). The total size of photos uploaded is 465 GB because the total number photos is 465,000, but for we will set the size to 1000 GB in case we need extra storage, so the price is $ 23.00(0.023 \* 1000). The number of requests are now 465,000 for each type of request, and the calculation unit is 465 (465,000 / 1,000) for PUT/POST request and 46.5 (465,000 / 10,000) for GET request. The prices for PUT/POST request and GET request are $ 2.33 (465 \* 0.005) and $ 0.19 (46.5 \* 0.004). Thus, the total price for S3 bucket is **$25.52**.

As for the cost of Amazon Rekognition API, we now need to process 465,000 pictures, so the price will be **$465** (465,000 / 1,000 \* $1).

As for the AWS Lambda function, the number of requests we will sent is 465,000, which is still below the 1 million standard, which means we can use it for free.

As for the cost of Amazon DynamoDB, the number of Provisioned Throughput for writes and reads are 465,000, which is below both the 2.5 million and 5.2 million tiers. The total size Indexed Data Storage is now 42 MB, which is still below 1 GB. So, the total cost will still be **$ 0.81**. However, the price for writes and reads Provisioned Throughput increases differently. The number of patients covered is calculated by dividing the number of writes/read with the total number of writes/reads per patients. The current unit price will cover about 2,500 patients (2,500,000/ (31\*15\*2)) for writes and 5,000 patients for reads (2,500,000 / (31\*15\*2)). We need to adjust the cost estimation accordingly for further increase.

The cost for Amazon EC2 instance stay the same because use the same type of instance for the same period of time. So, the cost is **$8.93**.

As for the maintenance cost, we estimate that the system needs one software engineer in Pittsburgh to work three days per month, which is **$ 702**.

Overall, the total price for an increment of 1000 user deploying our system is **$1,202.26**.

### *Increments of 1 Million Users*

Finally, we now increment our users to 1,000,000. The price for AWS S3 bucket will increase because the storage and the number of requests both increase (***Appendix 11***). The storage needed for now would be 465 TB (465 GB \* 1,000), and the price will be $ 10,281.2(50,000 \* 0.023 + 415,000 \* 0.022). The number of requests are now 465,000,000 for each type of request, and the calculation unit is 465,000 (465,000 / 1,000) for PUT/POST request and 46,500 (465,000,000 / 10,000) for GET request. The prices for PUT/POST request and GET request are $ 2,325 (465,000 \* 0.005) and $ 186 (46,500 \* 0.004). Thus, the total price for S3 bucket is **$ 12,792.2**.

As for the cost of Amazon Rekognition API, we now need to process 465 million (465,000 \* 1,000) pictures, so the price will be **$ 208.2** (1 \* $1 + 9 \* $ 0.8 + 90 \* $ 0.6 + 365 \* $ 0.4).

As for the AWS Lambda function, the number of requests we will sent is 465 million (465,000 \* 1,000), which is more than the 1 million free tier, and the total price will be **$ 92.8** ((465 - 1) \* 0.2).

As for the cost of Amazon DynamoDB, the number of Provisioned Throughput for writes and reads are 465,000,000 (465,000 \* 1,000). The Provisioned Throughput writes requires 186 write capacity unit (WCU) (465 / 2.5), which leads to the price of $ 87.42 (186 \* 0.47). On the other hand, the Provisioned Throughput writes requires 90 read capacity unit(RCU) (465 / 5.2), which leads to the price of $ 8.1. The total size Indexed Data Storage is now 42 GB (42 MB\* 1000), and the price will raise to $ 10.5 (42 \* 0.25). So, the total price for DynamoDB is **$ 106.02**.

Because of the increasing number of users, we decide to launch two instances in N Virginia region (us-east) and N California (us-west) region. The cost for Amazon EC2 instance is $8.93for N Virginia region, as mentioned before, and $ 11.16 for N California region. The total cost for EC2 instance is **$ 20.09**.

As for the maintenance cost, we scale up by 1,000, which is **$ 702,000**.

Overall, the total cost for 1,000,000 users is **$ 715,219.31**. And detailed comparison of these three scenarios are shown in ***Appendix 12***.

# Limitations of the System

In order to optimize the cost, the system extracts information from images using Amazon’s Rekognition. This is based on the fact that through a previous test on 200 images. The test shows that Rekognition and GPI(Google) have similar accuracy while Azure performs the worst. Considering that GPI cost much more money and cannot support images that are larger than 4 MB, the system is designed only with Rekognition. To further improve accuracy, it is better if the system can test and take the results from more machine learning APIs together to make a better decision.

To calculate the score for an image, the method for the system is to multiply score and confidence for each food item in it, sum them up and divide it by the sum of all the confidence. In order to achieve this, existing food labels and their unique scores should be stored for reference. Currently this is done manually. In order to improve the accuracy or robustness of the system, a better procedure is to use machine learning techniques to extract all the food labels and their unique scores.

Currently the server is flask server. A stress test for this server using *siege* is shown in ***Appendix 13.*** The result for the stress test shows that a flask server can deal with 780 transactions in a second. There are many currently popular web frames like bottle, tornado and django. Usually Bottle, if well designed, will have better performance than flask. So, the performance of the server may be improved by using another framework. Moreover, currently in the system Lambda Function and server are all implemented in Python. Python is not a very fast language. So, if the system can be implemented in other lower level languages like Java or C, the performance of the whole system will improve.

As there are not that many users, the System now uses only one instance. A safer consideration is to launch more instances located in different zone. Thus, if there is a damage on one of the instances, the service will not break down but will probably be able to be recovered.

# Scale-up Possibility

In the long run, our system is able to scale up to 100,000 users. 100,000 users may bring a large amount of application traffic, sometimes even simultaneously. To achieve this goal, firstly, we selected EC2 micro instance which is well enough to handle 1 million http requests per month. Meanwhile, we have made use of Elastic Load Balancing (ELB). It automatically distributes incoming application traffic across multiple targets, such as Amazon EC2 instances. Among all the three types of ELB, we adopted Application Load Balance. Application Load Balancer is best suited for load balancing of HTTP and HTTPS traffic and provides advanced request routing targeted at the destination EC2 that features the high availability, automatic scaling, and robust security to make our web application fault tolerant.

# https://lh4.googleusercontent.com/AbV3vSr7iempgf4OEHfRPAYvTiT68iMy6WdGaswKKgHPU0MOtae07lieGJ0biDUK3uI9KGWRM3BLHdVmgHl1Z3CD_Dr1Eo3vHxdxirzpaeI-I6hKYtCna29dYq7Iqzd5vyjGcRsxAppendix 1

# Appendix 2



# 

# Appendix 3

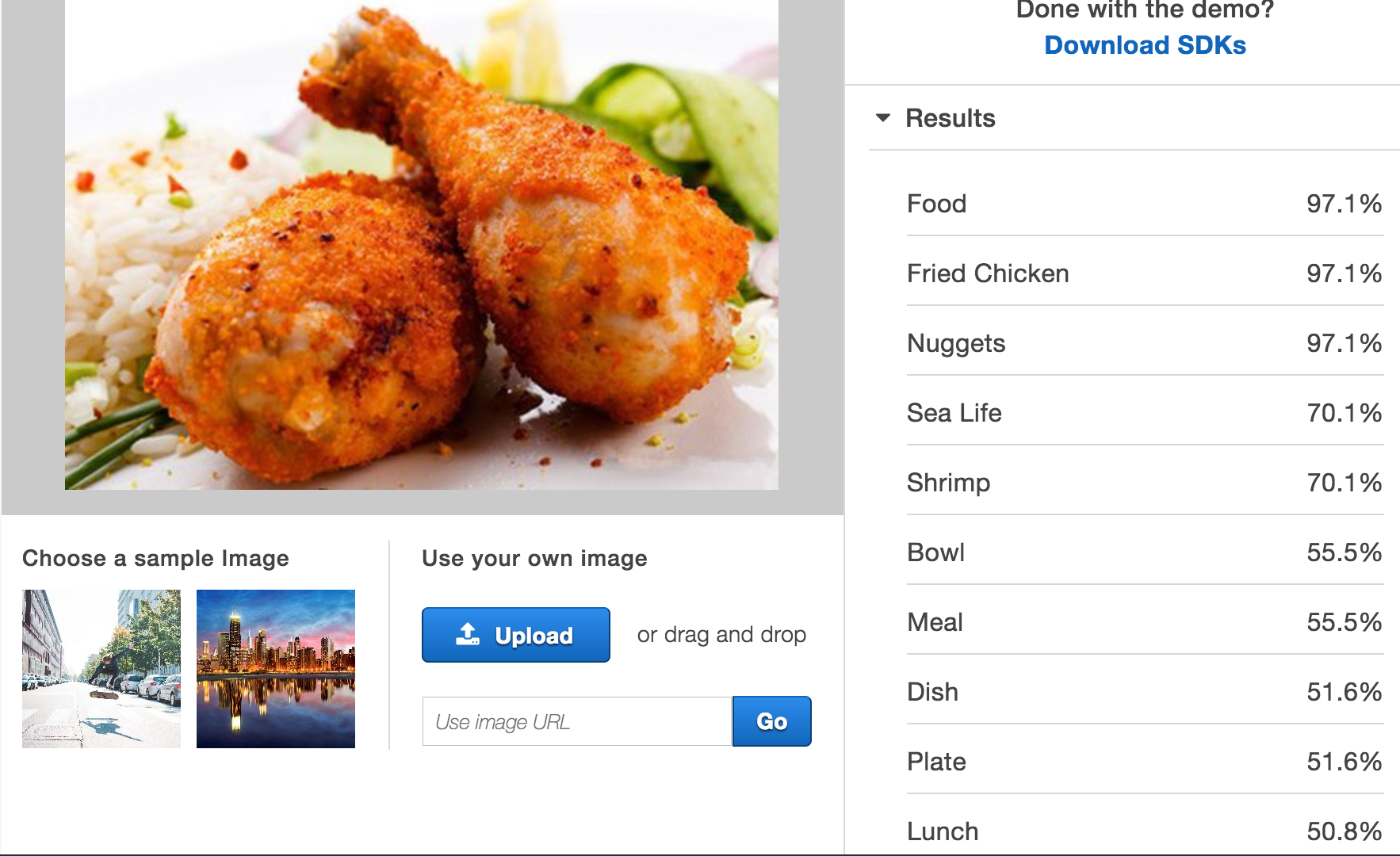
# 

# Appendix 4

# 

# Appendix 5

# Appendix 6



# Appendix 7

# Screen Shot 2017-09-25 at 5.46.19 PM.pngAppendix 8

# Appendix 9

# Screen Shot 2017-09-23 at 3.35.06 PM.pngAppendix 10

# 

# Screen Shot 2017-09-26 at 7.48.02 PM.pngAppendix 11

# 

# Appendix 12

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| The number of users | S3 Bucket  ($) | Rekognition API  ($) | Lambda Function  ($) | DynamoDB  ($) | EC2 Instance  ($) | Maintenance  ($) | Total Cost  ($) |
| 1 | 0.03 | 1.00 | 0.00 | 0.81 | 8.93 | 0.00 | **10.77** |
| 1000 | 25.52 | 465 | 0.00 | 0.81 | 8.93 | 702 | **1,202.26** |
| 1,000,000 | 12,792.20 | 208.20 | 92.80 | 106.02 | 20.09 | 702,000 | **715,219.31** |

# 

# Appendix 13

# 

# Appendix 14: Design Document

## A detailed description of the system and needed functionality

In our design, users upload the photo of food that a user eat for every meal. After that, when a user submits his/her user ID to the website that we designed. This user can easily attain a diet evaluation report including pie chart and point chart for him/her to grade and analysis the dietary habit of user. This user is able to down the month and annual analysis report for later use.

The whole process of our cloud computing design starts at uploading the photos of food. All of the photos will be uploaded S3 bucket on AWS (Amazon Web Services). Since we want use Lambda to run code and grade the food on the photo, SNS (Amazon Simple Notification Service) is a great bridge between S3 bucket and Lambda. When S3 bucket sends request to SNS, SNS will notify and send an event to Lambda. There are plenty of information in this event, such as photos’ keys. After lambda get the notification, lambda will execute the code by calling recognition method in API to recognize photos that we uploaded to the S3 bucket. All the type of the food on the photo will be label out after this. We give label a certain score and weighted out for every food on the photo by our algorithm. The total score of the photo has been calculated, whenever each step has been done correctly. Those scores of photos will be pushed to the Dynamodb to store them up.

When a user wants to check their dietary report, he/her is able to login on the website that we designed. Once submitted user’s ID through the website. The server on the EC2(Elastic Compute Cloud) will post user’s ID to the Dynamodb. On the Dynamodb, user’s ID will be match out and push the user’s daily meal score back to server and exhibit them on the website

## A detailed list of the system requirements

1. EC2 instance is required to keep running all the time.

2. All Images that needs to be uploaded must be “JPEG” and “PNG” format.

3. Name of image cannot contain “+” sign.

4. Lambda function response time/memory must be 1 min/512MB or higher.

## Assumptions made for the simulated system:

We have made several assumptions for our simulated system to facilitate our performance testing while also best imitating the real case scenario.

The first assumption is that all images in the simulated system are stored and uploaded through the S3 bucket for further analysis. In our simulations, test images are first stored in a public S3 bucket. Our system will then pull these images from the public bucket and copy them to another private S3 bucket. After this, the private S3 bucket will notify SNS of the images uploading event the SNS can then trigger the Lambda function to call the Amazon Rekognition API to analyze these images.

Another assumption is that our current simulated system focuses on analyzing the images and evaluating the healthy diet score for one single user. In our simulations, we assume the images uploaded belong to one user and we then assign the healthy diet score for him based on the image inputs.

The third assumption in our simulated system is that we take a general approach when evaluating the healthy diet score for the user. We assume a total of 100 points for the healthy diet scores and divide all foods into three categories: unhealthy foods, healthy foods and other foods that are fair. In our simulated system, we assume healthy foods mainly include vegetables, fruits, salads and other foods that are regarded as healthy by the general public; while unhealthy foods include deep-fried foods, desserts and other junk foods. A detailed explanation of our evaluation algorithms can be found in our report.

## An architectural design of the proposed system

The architectural design of the system can be described as two parts. The first part includes the input S3 Bucket, SNS, Lambda Function and DynamoDB, the second part consists of EC2 Instance and DynamoDB.

For the first part, the workflow goes as follows. When a client is taking food, the food he is eating will be captured by a camera and uploaded to the S3 Bucket. When an object is created (a new image appears in the S3 Bucket), the bucket will notify SNS. SNS will publish this event and trigger the specified Lambda Function. The Lambda Function extracts the information it needs from the event and grab the new uploaded image into its own space. When the image is downloaded, Lambda Function will call Rekognition API to analyze the features in the food. Lambda will calculate a score that indicates the healthy scale for the food in the image and store these information into DynamoDB.

For the second part, the EC2 Instance serves as a server. When user types in the URL in the web browser, the server will return a webpage in the user’s browser. The user type in the user ID, and when the server receives it, it will fetch the corresponding data from DynamoDB for this user. After doing relevant statistical work, the server will send the results back to the user, which will be user-friendly to view.

We will elaborate how S3 Bucket, SNS and Lambda Function works as a whole.  S3 Bucket is the place to store images. In this system, the S3 Bucket is configured to detect whether there is a new object created in it. When this happens, it will notify SNS. SNS is a service to publish events. In this system, SNS is configured to grant the very S3 Bucket to publish events through it. Also, it is configured that information about such event will be sent to the designated Lambda Function. For the Lambda Function, it is configured that it can be triggered by SNS so that it will react to the publish behavior from SNS. As in this system Lambda Function will grab images from S3 Bucket, it is authorized the right to do this. The image should be downloaded to the ‘/tmp’ path of the Lambda Function as this is the only directory that users customized function can take operations. Then Lambda will call Rekognition api to analyze the image. Rekognition will extract the items in the image and how much confidence that it can put into it. Forwardly Lambda will calculate the score for this image. It works as, sum up all the products of each food item, the product is obtained by multiplying the score of the item and its confidence, and then the result of the product divided by the sum of all the confidences will be the score for the image. The higher the score is, the healthier the food will be. When the score is retrieved, Lambda will insert the results including the score, time and a revised name into DynamoDB. The time is the time when this insert is done. The revised name is generated by adding a prefix to the original image name. This prefix is the user ID corresponding to the image and a number. The number is the total number of records that this user already has. In this way, images submitted at the same time or with the same name will also be successfully added.

## Tools, technologies and cloud services needed & evaluation

Overall, we utilize the following types of Amazon Web Services:

1. Simple Storage Service(S3) for image storage

2. Elastic Compute Cloud(EC2) functions as virtual server in the cloud

3. DynamoDB for database

4. Simple Notification Service(SNS)

5. Lambda function to receive image data and analysis images using Amazon Rekognition API

Below, we will evaluate why we choose Amazon AWS, Lambda Function, Amazon Rekognition API, DynamoDB and Flask.

* Amazon AWS

In terms of cloud computing services, we compared three service provides, namely Amazon AWS, Google GCP and Microsoft Azure. We first look through the cloud products offered by the three companies and conclude that Amazon provides the most comprehensive cloud services. Then we compare the pricing of their cloud services and discover that Amazon’s web services are offered at lower prices than Microsoft Azure but higher than Google GCP. Taking into consideration that Amazon provide more comprehensive cloud services, we believe that Amazon’s web services have a high cost performance ratio and therefore decide to deploy our system on Amazon Web Services.

1. Services Provided

In terms of user adoption, Amazon Web Service is the clear market leader, having cornered nearly half of the Infrastructure-as-a-Service (IaaS) market. With its first-mover advantage and nearly 5 years head-start, AWS offers a lot more cloud products and options. In contrast, Microsoft Azure and Google GCP is fairly new to the scene, and although it offers comparable solutions, it still lags behind. For instance, AWS offers a “serverless” compute product called AWS Lambda, which allows us to run code on-the-fly without having a dedicated instance waiting around for requests. This Lambda function is very important in our system design since the major part of our implementation is deployed on Lambda. However, Google’s serverless functions are still in Beta release stage and only supports serverless functions written in Node.js. while Microsoft Azure’s serverless functions are still not mature enough.

1. Pricing Consideration

Amazon AWS and Microsoft Azure ties with each other while Google GCP’s services offer lower prices. In terms of setting up instances, the on-demand hour cost for an Amazon t2.micro with 1 vCPU and 1 GB memory is $0.012 per hour. For Microsoft Azure, the cost for setting up a similar instance is $0.02 per hour. Google offers $0.0053/hour for setting up an instance with 1vCPU and 3.75GB memory. As for cloud storage costs, Amazon AWS’s regional storage costs are 2.3 cents/GB/month, Microsoft Azure’s storage services are 2.4 cents/GB/month while GCP’s regional storage only costs 2 cents/GB/month. Therefore, we could see Amazon web services slightly beat Microsoft Azure in price but are still more expensive than that of Google GCP.

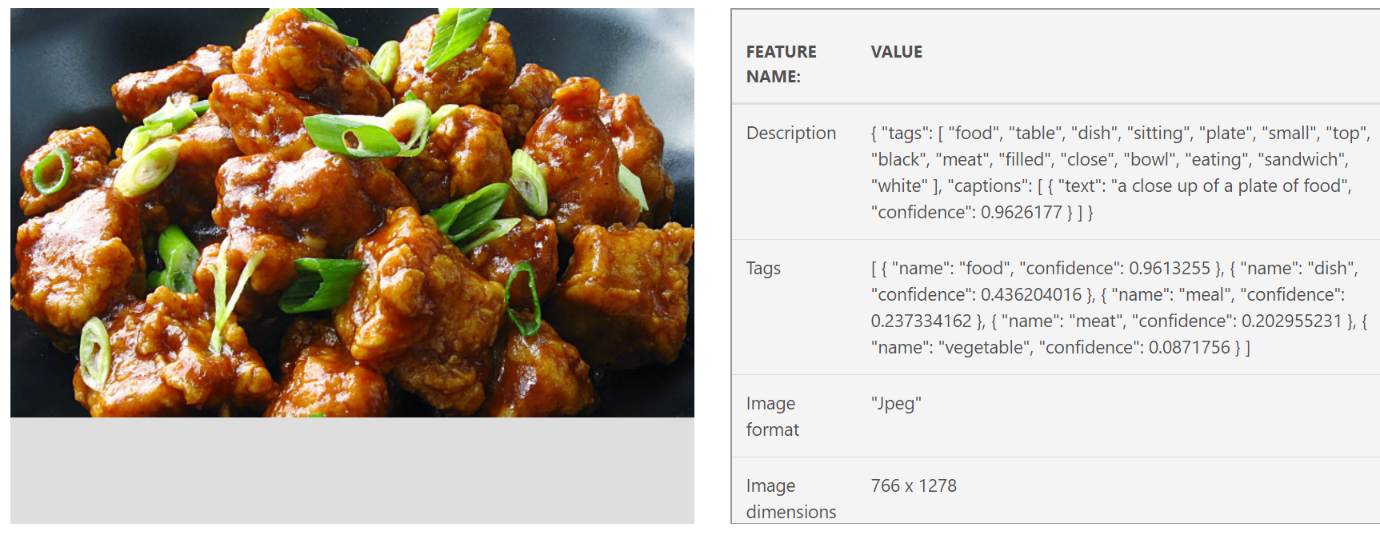
* Lambda Function

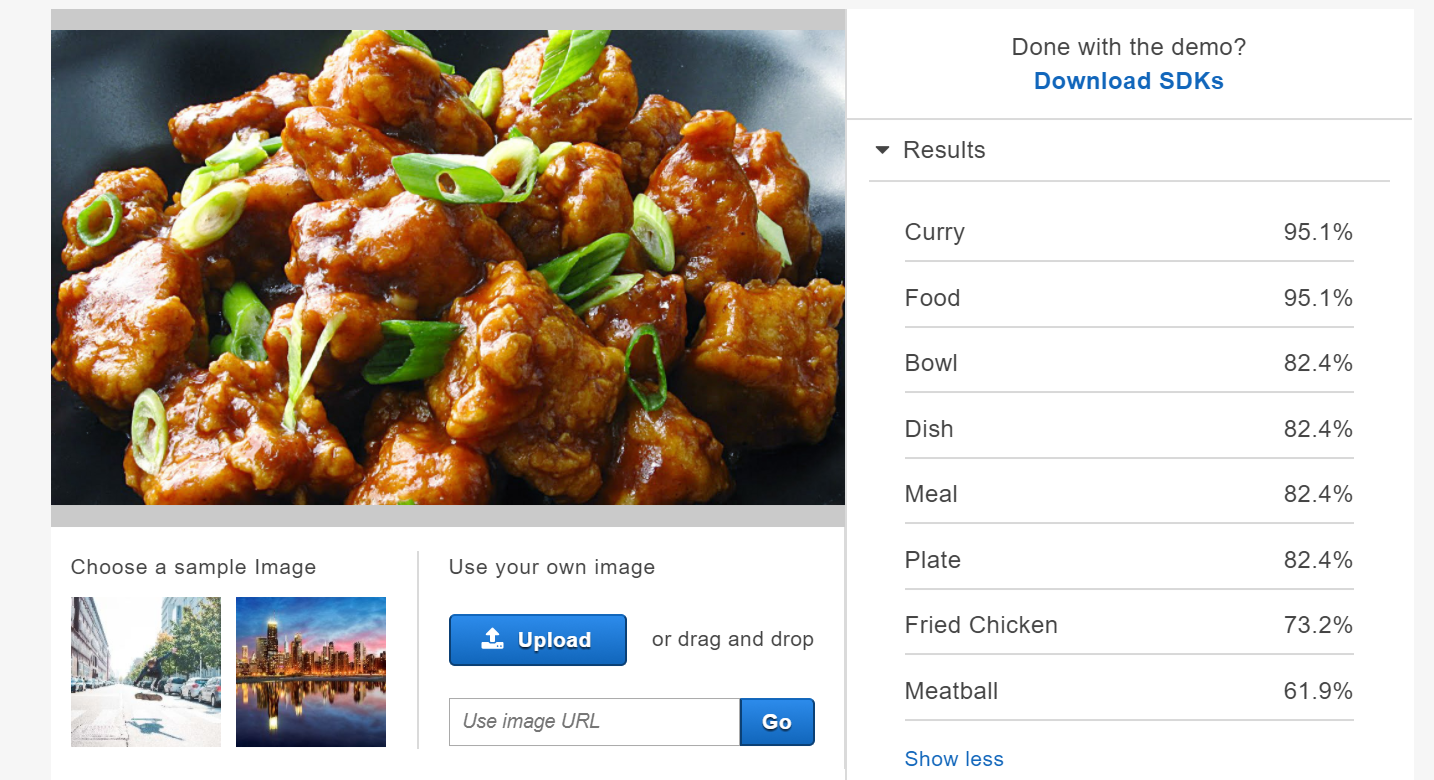
The reason why we choose Lambda Function is that when using Lambda, we only need to focus on our code. AWS Lambda manages the compute fleet that offers a balance of memory, CPU, network, and other resources. Unlike applications that are deployed to virtual machines or container runtimes, scaling for Lambda functions is handled dynamically in response to changing volume of function invocations, resources are allocated to the function and the customer pays for only the amount of resources uses. Cost is also cheaper if using Lambda Function. The billing for cloud functions is based on per invocation so it enables developers to reach a higher utilization of resources. In comparison, virtual machine incurs a cost even while idle.

* Amazon Rekognition

In terms of Image recognition, we compared three APIs, namely, Amazon Rekognition,

Google Cloud Vision API and Microsoft Azure Computer Vision API. All of them can detect object and extract features(labels) from images. We first ruled out Microsoft Azure Computer Vision API as it performed the worst accuracy after ten-image test. For example, for this typical picture of General Tso’s Chicken.



Food, dish and meal label is not correct but far from specific. The specific label ‘meat’ only has a confidence of about 0.20 which is not acceptable for use.

On the contrary, Amazon Rekognition has a confidence of 73.2% for label ‘Fried Chicken’ which is more convincing.

We finally chose Amazon Rekognition for its best support for AWS and its high analysis accuracy. The comparison of analysis accuracy between Amazon Rekognition and Google Cloud Vision are as follows:

1. First, we made a sample of 200 food images that cover a variety of categories, with different image size/format/resolution and with sources from real world and websites
2. Only labels above 50% confidence were recorded.
3. Based on our samples, 91% of labels detected by Google Cloud Vision API are correct or relevant while the percentage for Amazon Rekognition is 88%. More specifically, if the food image detected are taken by ourselves, the accuracy rate is on the same level. This indicates that the two APIs share similar performance because all the images uploaded in our system are from real world.

Meanwhile, we take pricing into consideration as it is a vital factor for our system.

1. Suppose we have 1000 users in the short term, and each user will upload up to fifteen images per day. Thus, there will be 450,000 images need to be processed per month.
2. For Amazon Rekognition API, in US-East (N. Virginia), for the first 1 million images processed, price for per 1,000 images processed is $1.00, so the total price is $450 per month.
3. For Google Cloud Vision API, the first 1000 images processed are for free, the 1001 to 5,000,000 images processed are $1.50 per 1000 images, so the total price is $673.5 per month.

If users of our system continue to rise in the long run, Google Cloud Vision API will definitely charge much more compared with Amazon Rekognition. Additionally, Google Cloud Vision API supports JPEG, PNG (both of which are supported in Amazon Rekognition) and other formats while failed to support images that are in excess of 4 MB. This is intolerable because many pictures taken by cameras are over 4 MB and need to be compressed.

* DynamoDB

DynamoDB is a typical NoSQL database, it excels in scaling, which, on the opposite, is need with effort in traditional relational database. Besides, DynamoDB is designed for failure (i.e., it has self-recovery and resilience built in), increasing the secure level of database. This is the primary reasons we choose DynamoDB because speed and scalability are big concerns for our system. Also, Amazon Web Service provides best support for DynamoDB which means it is easier to implement and deploy.

* Flask

Besides, we build our web application (for the purpose of displaying user’s diet data) using Flask on EC2 instance and utilize Highcharts.js library for data visualization. On the EC2 instance, there is a flask server. There are four popular web frames: bottle, flask, tornado and django. Bottle and flask perform better. The performance of bottle relies largely on the developer and can be really bad if not well developed. So, for this system a flask server is built. When this server receives a GET request, it will return an index web page that users can enter a user ID. In that page, user will be able to enter the ID and use the confirm button to send a POST request. After receiving this POST request, if the ID is not valid, the server will send back an error page. The error page will redirect the user to the index page in 3 seconds. If the ID is valid, the server will connect fetch out the data for this user and send back a web page with these data displayed in a user-friendly format - average score, historical data, and sources for each image. The sources will be displayed in a table, where their scores are listed, or indicating that the image is not about “food”.

## Realistic scenarios of end-to-end test

All the users of our system will carry a small camera with them. When a user consumes the food (including breakfast, lunch, dinner and snacks), the camera will automatically take a picture of the food and upload it into the public AWS S3 bucket to store the picture. We estimate that in the realistic scenario, the user will upload 15 images per day (three meals and snacks). We also use this estimate in our simulated system. Our system will transfer the picture from the public AWS S3 bucket to our AWS S3 buckets. After the transfer is completed, our system will calculate the healthy score of the food based on the picture uploaded by the user previously. The score will be stored in a database. The users can have access to their scoring details by logging in with their user ID on the website. The users can track their total score on all the food they consumed so far and their score for each food item (including time) at the same time. UPMC can also access to this information and use it to determine insurance quote.

# References

Amazon S3. (n.d.).https://aws.amazon.com/s3/

Amazon Dynamodb. (n.d.).https://aws.amazon.com/dynamodb/

Amazon Rekognition. (n.d.).https://aws.amazon.com/rekognition/

Amazon EC2. (n.d.).https://aws.amazon.com/ec2/

Amazon Lambda. (n.d.).https://aws.amazon.com/lambda/

Amazon Elastic Load Balancing. (n.d.). https://aws.amazon.com/elasticloadbalancing/

Flask. (n.d.). http://flask.pocoo.org/

Python common library. (n.d.). http://www.vimer.cn/archives/2926.html

Software Engineer Salaries in Pittsburgh (2017 Sep, 22) https://www.indeed.com/salaries/Software-Engineer-Salaries,-Pittsburgh-PA

AWS Simple Monthly Calculator. (n.d.). http://calculator.s3.amazonaws.com/index.html

Pricing Calculator Azure. (n.d.). https://azure.microsoft.com/engb/pricing/calculator/?ranMID=24542&ranEAID=TnL5HPStwNw&ranSiteID=TnL5HPStwNwh2XojOjvfPNOCbEYzsni9w&tduid=(f5894f1ed0f5f53d069c71805eae1817)(256380)(2459594)(TnL5HPStwNw-h2XojOjvfPNOCbEYzsni9w)()

Google cloud platform pricing calculator. (n.d.).

https://cloud.google.com/products/calculator/#id=799dd79f-132f-47f2-a5ec-27efc1732931

Robert Aboukhalil. (2017, Mar 13). Tale of two clouds: Amazon vs. Google

https://medium.com/@robaboukhalil/a-tale-of-two-clouds-amazon-vs-google-4f2520516a38