Nutrition Assistant Final Report

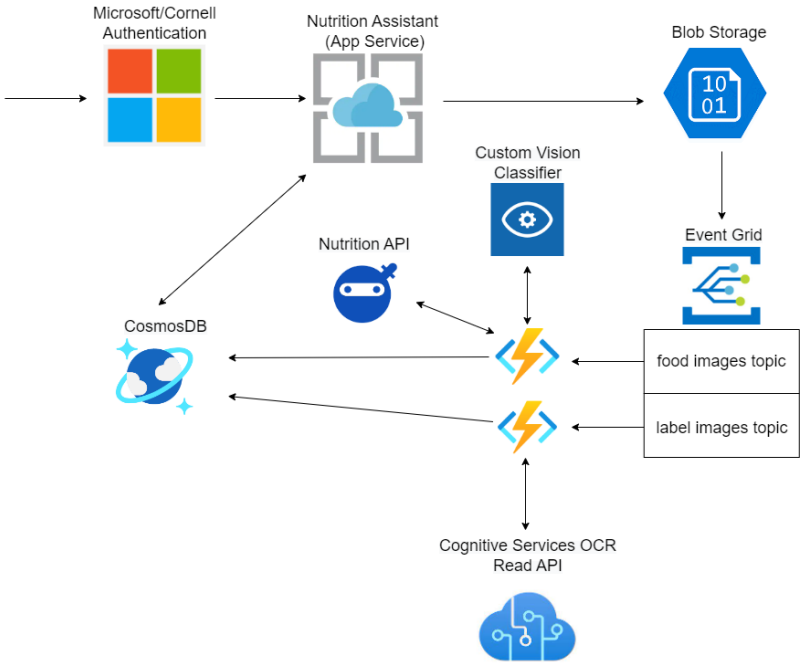
Berk Erdal, *be236@cornell.edu*
Morgan Cupp, *mmc274@cornell.edu*

Introduction

Tens of millions of people worldwide struggle with their diet every day, typically due to an imbalance or excess in macronutrient (carbohydrates, proteins, and fats) intake. According to the CDC[1], 39.8% of adults and 18.5% of youth in the United States struggled with obesity between 2015 and 2016. It is incredibly difficult to mentally keep track of one's caloric and nutrient intake throughout the day. Through this project, our goal was to help individuals easily track their food intake and make more informed eating decisions. Specifically, we wanted our system to extract nutrient information from (1) nutrition labels using optical character recognition (OCR) and (2) images of food items using a multi-class image classification model. Nutrient intake data would then be aggregated over time and accessed via a web application.

Popular products already in this space include applications such as MyFitnessPal and Lifesum. They allow users to log their food intake by scanning a barcode or taking a picture of their meal, while also providing healthy recipes, step tracking, and exercise logging features. However, many of these features require users to sign up for a premium plan with a monthly fee. We set out to build cloud infrastructure simulating food logging and nutrient aggregation to empower healthy eating habits with no monthly subscription, and we have achieved this goal. This report builds off of the intermediate report, incorporating system performance and scalability metrics, key development decisions, task division, and potential enhancements. Our application is available at https://nutrition-assistant.azurewebsites.net/.

Architecture and Technology



[1] https://www.cdc.gov/nchs/products/databriefs/db288.htm

Our application is deployed through the Azure App Service, providing seamless GitHub integration and configurable App Service Plan scalability. For security, we require users to login with their Microsoft/Cornell credentials. Users upload image data in two forms: nutrition labels or food items. Image preprocessing takes the form of file type checking and filename manipulation such that file names are prepended with a unique string identifier. Nutrition label uploads are directed to a *label-images* Blob Storage container, while food images are directed to a *food-images* container in a different storage account. Listening for blob creation events are two Azure Event Grid system topics *food-topic* and *label-topic*. Azure Blob Storage provides redundancy in the event that any service is overloaded or begins to fail, and routing events through Azure Event Grid allows us to separate the two key functions of our project into independent subsystems while providing excellent load balancing and scalability.

The Azure Function subscribed to the *label-topic* is responsible for optical character recognition. Given a Blob Storage image URI, this function makes an API call to an Azure Computer Vision OCR service endpoint. This service uses the generally available version of the API (version 3.2), which is available as a cloud service or Docker container. Access to the container image required filling out a special form and waiting for approval from the Cognitive Services team, so we opted for the cloud service form. From the service call response, this Azure Function extracts key macronutrient data (calories, fat, cholesterol, sodium, carbohydrates, fiber, sugar, and protein) and performs a write operation to the CosmosDB data store.

A separate Azure Function subscribed to the *food-topic* is responsible for multi-class image classification. We used Azure's Custom Vision service[2] to create, train, publish, and query an image classification model. This service provides a Python client library allowing us to efficiently tag and upload local training images, train multiple iterations of the model, and deploy the model to a predication endpoint. This Azure Function hits the prediction endpoint given a Blob Storage image URI, and returns a list of class predictions and their corresponding probabilities. Then, our function uses the API Ninjas Nutrition GET API[3] to get detailed nutrition information of the top prediction (assuming a standard serving size), and writes it to CosmosDB. We used the free tier of the API Ninjas service, granting us 50,000 API calls per month. Paid tiers grant up to five million API calls per month with data caching.

Lastly, CosmosDB provides a fully-managed NoSQL data store with fast read/write latencies, high availability, and usage-based elasticity. Nutrition data obtained from food images and labels are stored in a standardized format along with a timestamp. In the app, users may select a date range for which they would like to see their logged data. This generates two queries over the corresponding timestamp range. The first query sums the quantities of all macronutrients logged over this range, and the second query retrieves a list of classified food images with the respective confidence scores generated by our Custom Vision model.

Data Sources

Our project required two sources of image data: nutrition labels and food images. We used an online nutrition label generator[4] to create realistic, yet customizable nutrition labels and manually collected a small amount of nutrition label images as well. Generating easily verifiable macronutrient values proved useful when testing end-to-end system correctness. Furthermore, images collected manually were used to test system robustness on images of nutrition labels in different orientations and lighting conditions.

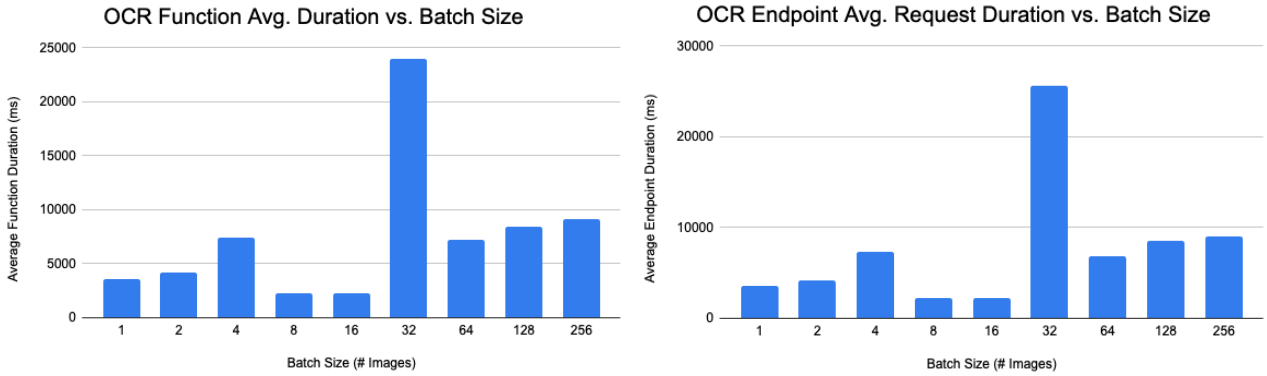
We used the Fruit Classification dataset from Kaggle[5] to train our food image classifier. This dataset contains 33 classes of fruits with a total of 22,495 images (16,854 for training, 5,641 for testing). With the Azure credit limit in mind, we implemented a smaller model that accurately classifies a subset of food items instead of a larger model spanning more food groups that is more expensive to train. We used a subset of the training data to train the model, and the rest of the training data and test data when simulating the data stream. Specifically, we used 33/50 image classes and 4224/5000 training images allowed by the free tier of the Azure Custom Vision's model training service.

[2]https://learn.microsoft.com/en-us/azure/cognitive-services/custom-vision-service/quickstarts/image-classification
[3]https://api-ninjas.com/api/nutrition
[4]https://www.onlinelabels.com/tools/nutrition-label-generator
[5]https://www.kaggle.com/datasets/sshikamaru/fruit-recognition

Performance and Scalability

Performance testing was done by uploading images of food items and nutrition labels (in batches of up to 256 at once) directly to their respective storage accounts. While our application lets users upload one image at a time, we believe this approach best simulates many users simultaneously uploading images and thereby system performance under load. The metrics for the following subsytems were collected by querying their corresponding Azure Metrics logs.

Function App - Optical Character Recognition

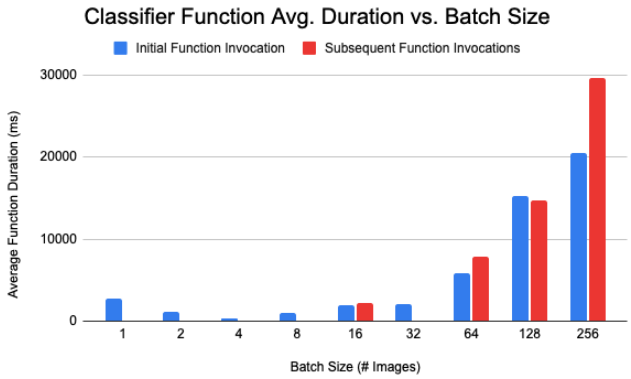


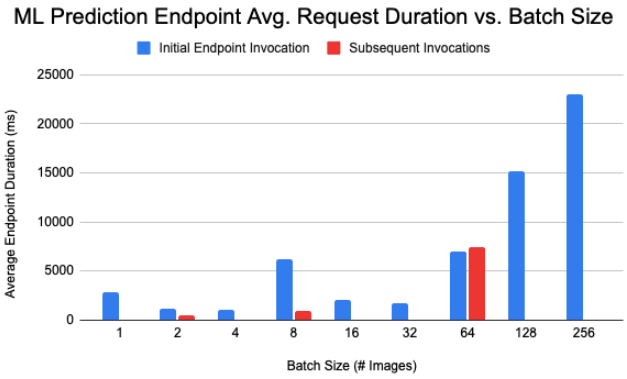
We tested the OCR function and endpoint by uploading nutrition label images in batches ranging from 1 to 256, the results of which are shown above. The graph on the left displays how the duration of the OCR function itself changes as batch size increases. Prior to a batch size of 32 images, the duration remains relatively flat with small fluctuations of $\sim$ 3000 ms. After 32 images, the duration appears to scale roughly linearly with the batch size. We initially chose the free tier of Cognitive Services' OCR plan, which provided 20 transactions per minute. However, due to scalability concerns, we eventually opted for the cheapest paid plan. This gave us 10 transactions per second (TPS) costing between \$0.40 - \$1 per 1000 transactions depending on usage. Consequently, batch sizes close to 10 images see little to no throttling, whereas batches larger than 10 images see a consistent, predictable decrease in performance. The right graph plots the duration of the OCR Read API endpoint against the batch size. This endpoint runs within the OCR function and is the function's main performance bottleneck, especially for large batches. This is evident since the relative durations on each graph at each point are very similar. Once again, the graph is flat prior to hitting the transaction threshold and begins scaling linearly after the threshold.

It is important to note the outlier on our graph with the batch size of 32. The duration of the endpoint, and hence the function itself, was over two times longer than the largest batch. Given the absence of service error logs, the additional overhead could be attributed to the Azure Function scale controller spawning a new function host[6]. Since the batch of 32 images was ran approximately 5 minutes after the batch of 16, In this case, Azure had reduced the number of instances to 0, requiring a cold start. We run the risk of cold starts because both of our Azure Functions are deployed on Consumption plans[7]. Function instances on the Consumption plan are scaled automatically, with each limited to 1.5 GB of memory, 1 CPU, and a function timeout of 5 minutes. Within the scope of this project, we found the Consumption plan to fit our needs. It granted us 1 million requests and 400,000 GBs of free resource consumption per month[8], allowing us to scale up to 200 concurrent functions without worrying about cost. Ultimately, this sort of delay could have been caused by numerous subsystems, including slow network speeds. In any case, all requests were eventually processed which demonstrates the fault-tolerance of our system.

[6] https://learn.microsoft.com/en-us/azure/azure-functions/event-driven-scaling
[7] https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale
[8] https://azure.microsoft.com/en-us/pricing/details/functions/

Function App - Machine Learning Prediction





In many ways, the data collected for the classifier function resembles that of the OCR function above. The average duration for both the function itself and the prediction endpoint stays flat while near the transaction limit. Once past the transaction limit, the request duration scaled roughly linearly, causing the function duration to increase. We once again conclude that the request duration to the prediction endpoint largely contributed to the function duration since the corresponding values for each batch were so similar. We initially used the free tier of Custom Vision's prediction service, which provided 2 TPS. This tier did not scale to our liking, so we upgraded to the cheapest paid plan providing 10 TPS for $2 per 1000 transactions.

The rate of linear scaling for larger batch sizes for machine learning prediction was notably more aggressive compared to OCR. Since both the OCR and Custom Vision classifier are on 10 TPS plans, we conclude that the Custom Vision endpoint tends to be slower than the OCR endpoint. The right graph supports this theory, as the endpoint duration is greater than the OCR endpoint's duration for large batch sizes. The other key difference in the data is represented by the red bars. When we collected classifier metrics, we found that the average function and/or endpoint durations were sometimes reported in two separate datapoints separated by less than a minute. As discussed above, we believe that this was likely the Azure functions host spinning up new instances in response to increased load. Thus, the red bars represent the duration of this "second round" of functions for a given batch. The OCR functions likely did not do this since the shorter duration of each endpoint meant the initial functions launched could handle all of the load at 10 TPS. In general, our functions scaled well and were only hindered by the TPS limits imposed by Azure. If this were a production-level tool generating revenue, we would purchase more expensive plan and achieve flat scaling up to a higher threshold.

The performance statistics for our model, as reported by Azure Custom Vision, were 99.4% precision and 95.1% recall. These are very high scores and indicate that our classifier's primary limitation is the subset of foods provided as training data. Our own testing also supports this idea.

CosmosDB

Azure CosmosDB provides extremely elastic scalability, 99.999% global read and write availability, and guaranteed read and write speeds of < 10 ms at the 99th percentile[9]. We opted for a serverless database model as opposed to a provisioned throughput model. A serverless model offers 50GB storage per container, < 10 ms read latency, and < 30 ms write latency[10]. Billing is done based on the number of request units, or RUs, used by our database operations. In our case, CosmosDB accounted for $0.20 in costs. This is opposed to a provisioned model where we would be charged for the provisioned RUs regardless of usage. At massive scale, we would choose a provisioned throughput model for better pricing, unlimited storage per container, and global region availability as opposed to our current single region availability. Regarding database consistency, we followed Azure's recommendation and chose session consistency, as it enforces monotonic read and write operations while offering high throughput and availability. Session consistency is the most widely used consistency level in single region and globally distributed applications.

[9] https://learn.microsoft.com/en-us/azure/cosmos-db/distribute-data-globally
[10] https://learn.microsoft.com/en-us/azure/cosmos-db/throughput-serverless

Other Subsystems

The App Service Plan powering our application runs on shared infrastructure on the Azure Cloud. With a cost of $10 dollars per month, this plan offers 1 GB memory and 240 minutes/day in compute, which exceeded our needs. However, this plan lacks certain features of the more expensive tiers, such as autoscaling capability, availability zone redundancy, and daily backups. Event Grid and Blob Storage provide reliable event delivery and secure object storage, respectively. Both services offer low latencies at massive scale, well beyond the needs of this application while providing a cost-effective, pay-as-you-go pricing model.

Key Development Decisions

In the planning/exploratory phase of this project, we sought to simulate food logging by streaming image data from an IoT Edge device (Linux VM) to the Azure Cloud through an IoT Hub. After TA feedback on the intermediate report, we realized that using a web application would make our service significantly more accessible and likely to be adopted by the public compared to a solution requiring special IoT hardware. So, we switched to a managed, scalable solution in the App Service by following this tutorial[11]. Here, we chose a Windows App Service plan over a Linux plan. While a Linux Service plan would have offered faster speeds and more customizability, we followed Azure's recommendation and opted for a Windows plan for seamless integration with other Azure services and expansive language support.

Additionally, when beginning this project, we were unaware of the various discrepancies between different standards/formats of nutrition labels. These include multiple columns, abbreviated macronutrient names, missing macronutrient values, among others. Accounting for these edge cases was hindering development, so we decided to target the format of the FDA's newest nutrition label[12]. Furthermore, we did not anticipate the nuanced manner in which the OCR Read API treated certain nutrition label fields. As a result, we learned the ins and outs about how the service handled nutrition labels, such the 'Calories' value being returned on a separate output line, and 'O' characters occasionally being interpreted as zeros.

Finally, our project originally had both storage containers under the umbrella of one storage account and one Event Grid topic. This was sufficient when testing locally, but after fully deploying our solution, we discovered that unpredictable event routing was causing our Azure Functions to trigger on both types of image inputs. The effects of this were especially noticeable at scale, where the number of function invocations and errors were suspiciously high. Switching to a model with two topics and two Blob Storage containers in separate storage accounts remedied this problem, increasing our throughput while reducing costs.

Task Division

Both group members were involved in the development of all components of the project, with the majority of the work being done through pair programming sessions after class. Initial brainstorming and implementation of the App Service, Blob Storage, OCR Function, and CosmosDB were done together. Berk focused more on implementing, training, and deploying the Custom Vision classifier. Morgan focused more on implementing the database queries, generating and displaying data tables, and handling errors in the frontend. Both Azure accounts were used to experiment during development, but we used Berk's Azure account for final deployment, performance and scalability testing, and tuning service configurations.

Future Work

We see numerous opportunities for this project to grow. First, our food classifier could be expanded to cover other food groups and our OCR function tuned to handle unorthodox nutrition label formats. While our solution handles the majority of formats, our decision to focus on the newest nutrition label standard left out some edge cases. Finally, our app could allow users to input custom food weights and/or serving sizes. Both Azure Functions assume the user has consumed one serving size of their respective food, but this is not very realistic. This is achievable as API Ninjas Nutrition API already supports custom portioning.

[11] https://learn.microsoft.com/en-us/azure/event-grid/storage-upload-process-images
[12] https://www.fda.gov/food/new-nutrition-facts-label/whats-new-nutrition-facts-label