Food Tracker

Jaeyoon Choi, Zhengze Gong, Keaton Drebes

Department of Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—A smart device that will automatically keep track of your kitchen's inventory for you. This project consists of one or more embedded devices that propagate information to a cloud server. The user can interact with a web application to view their current kitchen inventory, and/or create a shopping list based on recipes that the user has added.**

***Index Terms*— Classification, Cloud Computing, Computer Vision, Localization, OpenCV, SIFT, Smart Home, SSIM, Web-Application**

I. Introduction

In the hustle and bustle of the modern world, oftentimes it can be challenging to keep track of the exact contents of your kitchen when going grocery shopping once-weekly. This can lead to purchasing items you already have, or forgetting to purchase an item that you may need. The former leading to wasted food (if the item is perishable), while the latter leading to wasted time from returning to the store to buy any accidentally omitted items, or the creation of an imperfect meal, all relatively common occurrences for anyone who does any amount of home cooking. As such, we aim to provide a solution in the form of an embedded smart device.

At its core, our project is an inventory tracking system. It maintains an automatically generated inventory of items in the user’s kitchen using a Raspberry Pi (RPi) and an embedded camera, and cloud-side computer vision (CV). This hardware sits on the ceiling of a storage area like a fridge or a cabinet and captures images of the interior to send to a cloud computing server for processing.

Existing kitchen inventory solutions generally belong to two categories: focused exclusively on restaurant inventory management, and app-based tracking systems. Restaurant systems are not only costly and complicated, averaging $99 to $129 per month for point-of-sale integrated systems for small-business use cases, but also extremely excessive for the average consumer in a home kitchen [5]. App-based tracking systems on mobile devices, like Out of Milk and NoWaste, have the most similar use-case, but most (if not all) rely on the user manually inputting items, and quantities into the app with little to no automation. (If automated solutions exist, they have failed at informing and/or capturing the market.) As such, our system reduces the user burden by using computer vision to identify items and automatically cataloging them whenever possible.

II. Use-Case Requirements

Ease of use: Given our market niche, from a usability perspective our project should be as unobtrusive as possible. All in all the system should take a minimal amount of time to set up, minimal input from the user to operate, and minimal disruption to the user's normal routine when storing/retrieving groceries. To quantify these requirements, the total setup time, including account creation on the web-app and registering the sensor, should take less than five minutes to complete. This time amount is relatively arbitrary, but seems reasonable for what a user would expect of a smart device that only needs a one-time setup process. The system should update automatically in the background, and only notify the user in a limited handful of scenarios, adhering to the principle of minimal user disruption.

Multiple Users: The database should be able to handle multiple registered users, and multiple devices per user. Having this use-case requirement makes future scaling of our system much easier to implement.

Response time: Research suggests that most web-app users expect responses within 3 seconds [1]. Therefore, it would be good if the total response time of the system, from door close to web-app update, was less than three seconds. (See Section IV for further breakdown of expected response time values).

Item Handling: Our project should be able to handle both supported and unsupported grocery items. While we try to support a number of common grocery items, different users with diverse culinary preferences likely have wildly varying purchasing habits with regards to groceries. We’d like to ensure that our system is flexible enough to handle such cases without breaking or returning an error, while also ensuring manual item entry works as expected. The user is notified of any

TABLE I. Possible CV Outcomes

Items[a]	**CV Possible Outcomes**		
	ID’d as A	***ID’d as B***	***Fails to ID***
Supported item A	True positive → good ID 85%	False positive → bad ID 5%	False negative → failed ID 10%
Unsupported item	False positive → bad ID 5%		True negative → no ID (good) 95%

[a] Let X and Y be arbitrary supported items, A ≠ B

Fig. 1. Table breakdown of identification cases. Notice that each row should sum to 100%.

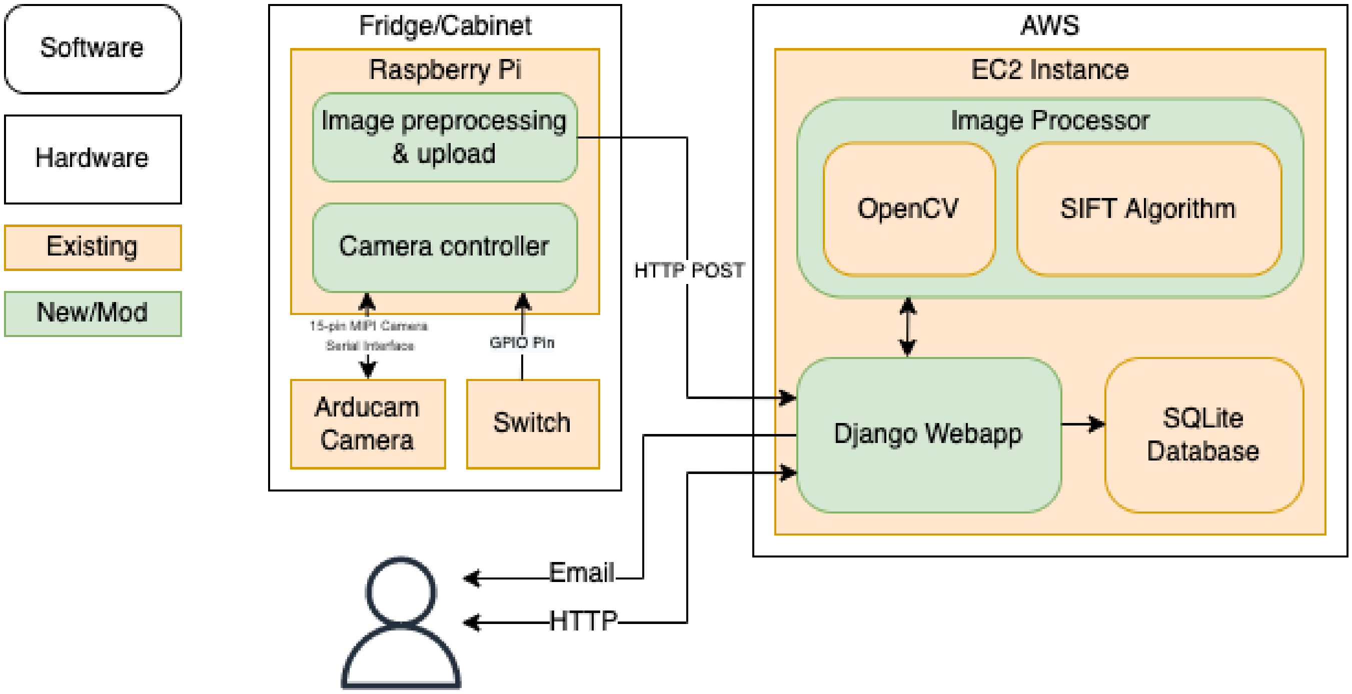


Fig. 2. System block diagram.

unsupported items, also pursuant to the earlier goal of minimal user disruption.

Erroneous Identification: Additionally, we would like to minimize the number of identification errors, which includes false positives and false negatives (see Fig. 1). It is especially important that we minimize *false positives* (incorrectly identifying one object as another), which would cause the greatest disruption in kitchen inventory assumptions. False positives are more likely to go unnoticed until the user reaches into their cabinet and notices the distinct *absence* of a key ingredient, and could lead users to either not purchase an item that they mistakenly believe they have, or double-stock an item that they do have but was misidentified. That is to say, we would much rather have a failed ID than a bad ID.

User Recipes: Users can add their own recipes to their personal cookbook, and the system automatically checks whether the user has all of the necessary ingredients for a given recipe. If there are any ingredients missing, the system automatically generates a grocery list, which the user can access. Relatedly, the system supports sending the user a grocery list with the said ingredients via both text and email. Users can also share their recipes and view recipes shared by others on a dedicated page.

III. Architecture and/or Principle of Operation

The project consists of one (or optionally more) hardware unit(s) that captures photos, a cloud server that handles the computer vision, and a web-app that presents the information to the user in a legible format (Fig. 2).

On a high level, the hardware component captures an image of the storage area when the user closes the door, represented by a 'button press' from a cabinet door closing, and sends it to a cloud server running our computer vision algorithm. The algorithm tries to identify the item added/removed by the user. If the item can be successfully identified, the server updates the user's inventory in the database. Otherwise, it notifies the user and asks them to label the item manually. The user can view their inventory or label unidentified items through a web application.

Our system has a few minor changes from those initially specified in the design report. Firstly, the camera was changed to be placed directly above the storage location as opposed to at a 45 degree angle, which allowed better captures of any item labeling. Additionally, we specified that the grocery lists can be sent via text and email (we weren't certain we would get to it, at time of writing the design doc).

A. Hardware

The hardware unit consists of an RPi attached to a wide-angle camera module, and a pushbutton switch. The camera is placed directly above the items in the storage location, at such a height as to ensure full view of the items. For the purpose of our demo, the hardware unit was attached by a flap to the top of a cardboard box (our storage unit).

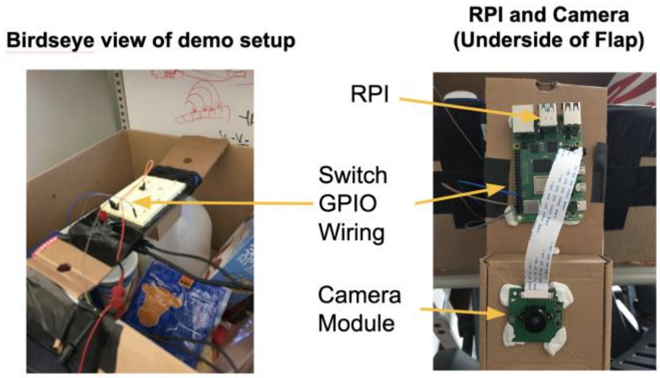

Birdseye view of demo setup
RPI and Camera (Underside of Flap)
RPI
Switch GPIO Wiring
Camera Module

Fig. 3. Labeled hardware unit.

B. Hardware to Cloud

The hardware component communicates with the cloud deployment with a POST request containing the device's ID, a serialized image, the serial number of the RPi, an integer timestamp, and a secret validation string. Each of the hardware units are given a secret validation string, so that a malicious actor can't impersonate another device and post fake JSON with just the serial number. Similarly, the timestamp exists so that the server can ignore repeated requests, if a malicious actor tries to repeatedly send a message (replay attack). For security reasons the POST requests are encrypted, and the server verifies the correctness of the string via its internal mapping using a JSON schema before processing the rest of the request.

C. Computer Vision Algorithm

From here, our cloud deployment of our CV algorithm identifies the item from the image sent. For the computer vision component, we use SIFT[3], as it was found to be the most effective of the algorithms which we tested. In addition to SIFT, we also use some secondary checks in order to increase the accuracy in cases known to cause confusion. (See Section V for a more comprehensive comparison of the potential advantages and disadvantages of the available algorithms).

D. Cloud Deployment

Both our CV algorithm and our web application (web-app) run on the same EC2 instance. Doing so cuts down on our cost of running two servers, while reducing our complexity in implementation. Furthermore, because SIFT is a comparatively lightweight algorithm that doesn't require much computing power, we felt it didn't warrant using dedicated GPU resources or running on multiple instances.

E. Models used in the Database

Within the scope of this project, we have several models used to store data. Of note beside the standard User, Device, and Recipe classes is the distinction between Category and ItemEntry. Categories describe general types of items. For example, Milk is an item Category. If a user adds a bottle of milk to a cabinet, then an ItemEntry instance with the Category of Milk is made for that specific bottle. This distinction is needed so that the CV code can classify objects correctly. This also allows users to create recipes without being beholden to the list of ingredients currently in their kitchens.

F. Web Application (Web-app)

As mentioned above, the web-app and the CV code run on the same EC2 instance. Users can log in, view their registered devices and inventories of those devices, add and modify recipes that they have stored, based on the ingredients the user is missing. The shopping lists can be sent to the user via either email or text. All of the supported items are stored in a database with quantity currently in the fridge and descriptors of the iconic images.

The system also notifies the user if/when an unsupported item is placed in the cabinet, by displaying a list of UNKNOWN ITEM's for each storage location. The user can then manually identify the item, tagging it as either an existing category of item (the 10 supported item classes by default), or register it as a new item category class. Should the user register the item as a new class, it can be detected in the same manner as the other registered items, but only for that user. For example, if user A registers say, duct tape, any duct tape stored by user B will not be identified as duct tape unless they also go through the process of registering it (see Section V for more information on why we do this).

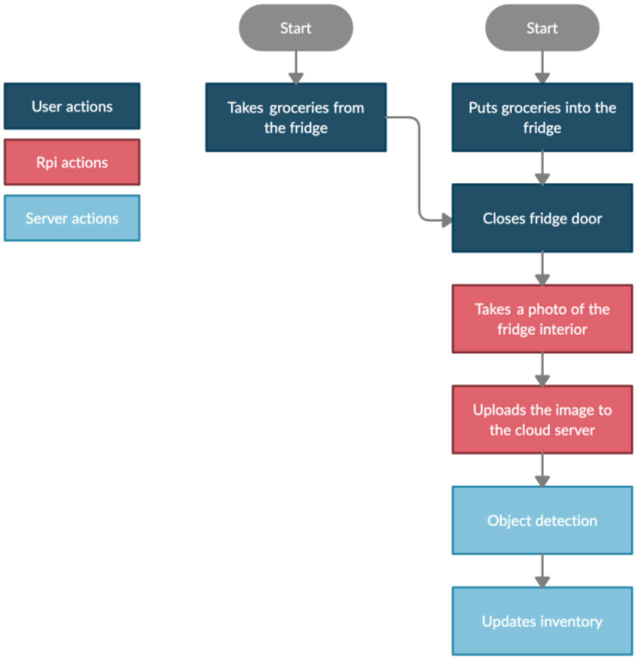

User actions
Rpi actions
Server actions
Start
Start
Takes groceries from the fridge
Puts groceries into the fridge
Closes fridge door
Takes a photo of the fridge interior
Uploads the image to the cloud server
Object detection
Updates inventory

Fig. 4. User action flow chart.

IV. Design Requirements

For our design requirements, most of the requirements we discussed in Section II can be carried forward into the final design requirements, provided we place a greater burden on the user than we would ideally want to. While this is unfortunate, we saw no other way if we wanted to keep this project within a reasonable scope.

The first requirement we place on the user is the requirement to only ***retrieve/store one item at a time***. While this is a hefty requirement, but it is needed to satisfy a different use case requirement, the ability to handle unsupported items. This enables us to perform SSIM differentiation (a type of pixel differentiation) to localize any object that we do not support. This also helps us with localization in general, which makes it easier to perform accurate classification.

The second requirement is that the user arranges objects within the storage location such that the ***label is visible to the camera***. Attempting to classify objects without the labeling being visible is simply not possible, so we must place this burden on the user. While this could be remedied by installing additional cameras, our use-case requirement of minimal user effort during installation and the ease of implementation ultimately led to this decision.

While researching, we found an example of using YOLO for object localization/classification with messy backgrounds for grocery items that managed to achieve ~85% mean average precision [2]. Given that we really don't have to handle localization at all, and we do classification with an uncluttered background, we feel that an ***85% accuracy rating*** is achievable for our use case. We also feel that this is a reasonable requirement, given our initial success with testing the various algorithms (see Section V).

For the time delay, while doing the tests to compare the various algorithms, we found that the keypoint/feature extraction with SIFT on a basic laptop took about .55 seconds per image on average, with worst case images taking about 4.5 seconds. The manually taken Apple Sauce and Crushed Tomato Sauce images taking about 4.5 seconds each. The feature matching took an average of .033 seconds, with a worst case of about 0.25 seconds. Again, most of the worst cases were comparisons with the manually taken Apple Sauce and Crushed Tomato Sauce images. At this time, we are uncertain of exactly why there is such a variance between the average and worst case times. Therefore, we decided to base our update time requirement on the worst case scenarios of the CV component; We allow for ***no more than 10 seconds from item placement to website update***.

Fig. 5. The item with the longest feature matching. The can of crushed tomatoes took about 4.5 seconds to extract features.

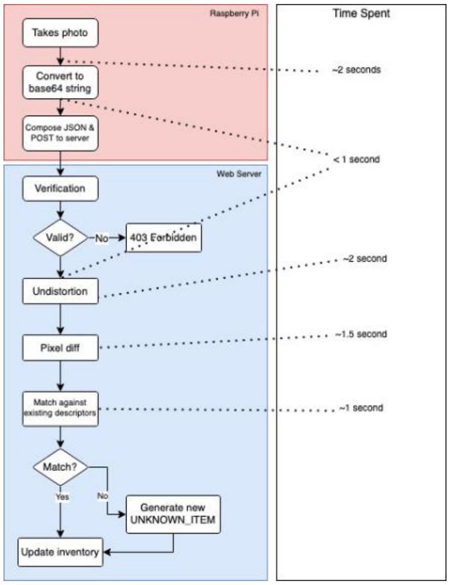


Fig. 6. Breakdown of approximate time spent for any given item.

V. Design Trade Studies

There were several major decisions we made while iterating on our design.

A. SQLite vs. Alternatives

SQLite is generally considered to be less effective when handling large numbers of transactions concurrently. For larger industrial deployments, MySQL could be a worthy investment for security and scalability, especially when considering that our main user-facing interactions happen on a web-app. However, it's clear that, at least for now, the costs outweigh the benefits. Aside from not having the capital or the infrastructure to run a database, our project is lightweight enough to run on an EC2 instance even with multiple users, and we anticipate each user not generating a large number of transactions. According to a study by a consumer group in the UK, people open a cabinet ~9 times a day. we estimate ~10 (non-heartbeat) transactions per user per day [4] on average, given how often people . Additionally, we are using Django for the web-app, which already supports SQLite as the default database backend. Therefore, we decided to use SQLite given the lack of performance concern and due to the time cost of switching to an alternative.

B. SIFT vs. Alternatives

When determining the algorithm to use for object detection and classification, we looked at SIFT, ORB, BRIEF, and YOLO for detection in conjunction with a CNN for classification. While we were doing our preliminary investigations, we found that the non neural network algorithms seemed to perform sufficiently well for our purposes (provided that the labeling was visible). While the neural network based classifiers had the potential to be better, we had no guarantee. Additionally, while we found several datasets of grocery images, none of them worked for our purposes, so the amount of effort that would be required to generate the required training data would have likely been substantial. Finally, if we used a neural network based algorithm, we would have been unable to register arbitrary grocery items on the fly, since we couldn't expect the user to provide enough training data to retrain the model every time they need to register a new grocery item. Therefore, we decided against using them, and instead focused on SIFT, ORB, and BRIEF, which already showed us tangibly successful results.

In order to do a simple benchmark of the three algorithms, we tested the effectiveness of the classification by measuring the quantity of total matches/good matches using Lowe's ratio test [8]. For the Iconic images, we used images taken from grocery store online catalogs. For the actual images, we took photos of the products, and manually cut out the background, to simulate the pixel differentiation that would take place. This does not perfectly simulate the photos taken from the sensor for a number of reasons: The final images have some lens distortion, and the focus is not as good. However, we feel that this test was effective for the purpose of comparing the given algorithms.

As shown in Table IV (Appendix), SIFT completely trounced the other algorithms. BRIEF misidentified 10/11 of the item classes which were tested, ORB failed slightly better, misidentifying 4/11 of the items tested. SIFT failed for only 2/11 item classes (Eggs and Milk). Additionally, SIFT tended to have a much greater difference between the correct/incorrect items. For BRIEF, the difference in positive feature matches between the identification with the most positive matches and all other identifications was, on average, 7.55%. This was 57.4% for ORB, and 76.4% for SIFT. This shows that SIFT is much better at distinguishing between items in the success cases.

C. Cloud Deployment vs. Jetson Nano

A popular device for performing CV computations is the Jetson Nano, whose powerful specs are more than capable for our use case. However, we decided not to use this specialized hardware for the following reasons.

- *Low utilization*: our use case has very low utilization because a user is only likely to move grocery items ~22 times a day [4]. Although we can get lower latency with local instantiation, the computational resources of the Jetson Nano would be wasted when the system is idle.
- *Lag tolerance of the system*: the lag penalty for doing the CV remotely is a fairly minor issue because we don't expect the user to check their inventory right after they put items into the storage area.
- *Miniscule data transfer*: our project already necessitates a network connection for the item list to be accessible anytime via the web-app. Since the data we are transferring over the network only consists of an image and some text fields, which is expected to be around 2~3MB, the user's network throughput is unlikely to pose an issue.
- *Physical Size*: adding a Jetson Nano to our hardware module would have required a larger physical footprint in terms of volume. As our goal is to make the hardware module as unobtrusive as possible, a Jetson Nano would make the user installation more inconvenient.
- *Cost*: a typical mini fridge costs around $150~$250 on Amazon, and a Jetson Nano kit is listed at $100+, which is more than half the price of some fridges. If our product were to be commercialized and scaled, the cost of a Jetson Nano alone could be a high barrier to entry that would dissuade many users.

Therefore, given how difficult we felt it was to justify the cost of having dedicated local hardware, and given how flexible our use case was to alternative solutions, we ultimately decided to do the computer vision processing remotely.

D. RPi vs. Alternatives

There were several possibilities for the embedded hardware we could use, Jetson Nano, Aduino, etc. Ideally, we would want the cheapest possible hardware that could take a photo, and send a post request to the web-app with the required information. However, we decided to use the RPi because it was simpler to prototype: in addition to being widely available, it has built-in wifi capabilities, a selection of camera modules that had a python package that allowed for easy control of the camera module, and ample documentation online for troubleshooting.

E. OAuth vs. Proprietary Account Management

Our use case requires us to maintain the inventory of several different users concurrently. Therefore, we need some method of handling login and password management. Our options were to utilize OAuth, or to store the information ourselves. OAuth provides superior security and user experience, as it is much more convenient than creating a separate username and password for this specific service. Additionally, we don't anticipate implementing OAuth to be any more difficult than handling it ourselves.

F. Global vs. Local Updates to Supported Items

To define Global and Local updates to supported items: A *global update* is when a user stores an item that is not supported, provides an iconic image when prompted, and the item is added to the global set of supported items for all users. A *local update* is when the user goes through the same process, but the item is only added to the user's own personal set of supported items.

Ultimately, we decided on local updates for two main reasons. Global updates to the supported items list would leave the CV algorithm vulnerable to potential griefing by a handful of malicious users who could intentionally mislabel common goods, or even label them with obscenities. Furthermore, Global updates would also result in additional image comparisons for every item in every user's inventory, even if they don't regularly use the item in question, significantly driving up response time and hindering user experience. Global updating does have the advantage that we won't have multiple users running into the same coverage gaps—perhaps this could be leveraged into a new feature in future improvements.

G. Django vs. Flask

Members of our team already had experience using Django, and there weren't any specific features offered by Flask which are needed for our use case. Therefore, we've decided to just use Django as the framework of choice.

H. Wide Angle Lens vs. Standard Lens

Initially, we wanted to use a wide angle lens in order to ensure that we had full vision of the items, regardless of how close the camera was to the items. The distance of the lens to the item ultimately was a non-issue, while the usage of a wide angle lens became slightly problematic due slightly incorrect undistortion (see Section VI). Additionally, due to replacing our original set of camera modules, we were unable to easily remove the lens. Ultimately, since the issues with distortion were minor, we decided to continue using the wide angle lens instead of purchasing another camera module.

VI. System Implementation

A. Hardware

The hardware component consists of a Raspberry Pi (RPi) model 4B, the OV5647 Arducam camera module, and a pushbutton wired to the RPi. While we initially specified an RPi 3B, as it is a cheaper model, our hardware requirements only

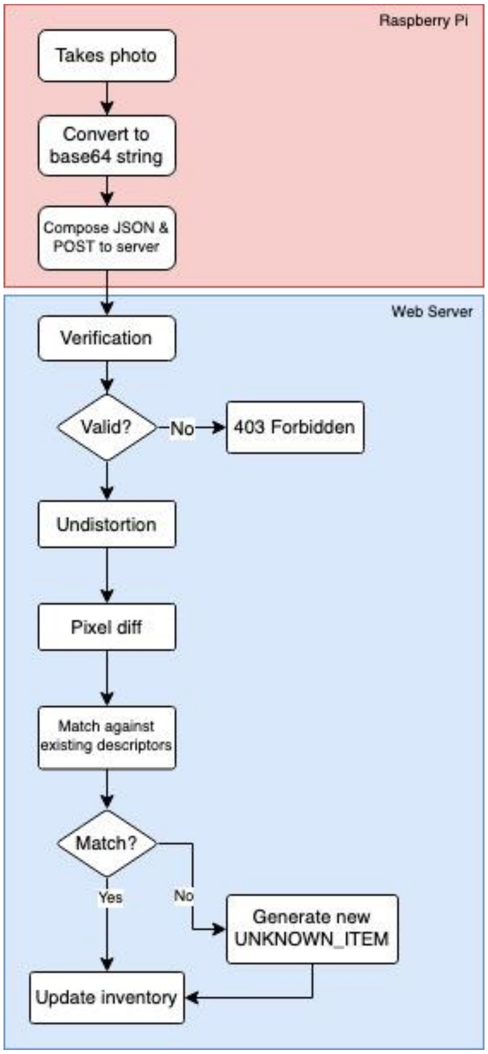


Fig. 7. Computer Vision flowchart.

require built-in wifi connectivity and a 15-pin MIPI Camera Serial Interface Type 2 (CSI-2) port for the camera hardware. Accordingly, when the CSI port on the 3B was accidentally broken, we replaced our design specification with the 4B as it was the model that could be delivered the fastest. Because of this hardware component, we do require that the RPi have some power supplied to it either in the form of a power bank or an outlet connection.

Our system also has a button connected to GPIO pin 5 of the RPi, and placed by the cabinet door to “sense” (read: simulate) door closure. While it’d be wonderful for a hypothetical real-world deployment to have an integrated sensor unit built into a smart storage appliance, having a simple button to detect cabinet closure also allows for retrofitting existing “dumb” appliances and standard cabinets with minimal installation, and in a way is more flexible and accessible to more people.

The OV5647 Arducam camera module connects directly to the RPi’s CSI port. This sensor works in the same manner as the native RPi camera module, meaning we can use the picam Python package to easily control it. The OV5647 comes pre-equipped with a wide angle M12 lens.

Each RPi knows its own serial ID, the number of messages it has sent to the web server, and its secret string, all of which are used when communicating with the web-app. The sensor sits idle until it detects a door closing (reads a button press on GPIO pin 5). When this happens the RPi takes a photo, and sends it to the web-app by an HTTP POST, before returning to idle. (See Section VI.C for details on the communication between the web-app and the hardware component.)

B. *Web-app*

The web-application is hosted on an AWS EC2 instance. The EC2 instance we used for the live demo was t3-2xlarge, which has 8 virtual CPU cores (vCPU) and 32 GB of memory. During testing, it was hosted on t2.xlarge, which has 4 vCPUs and 16 GB memory. While the upgrade wasn’t strictly necessary, we wanted to minimize the number of possible hiccups during the live demo, as well as demonstrate that scaling up using AWS is a seamless process. We use an Apache server in order to connect the Django backend to the incoming HTTP traffic.

We assigned an elastic IP on AWS for our EC2 instance so that the IPv4 address for the server stays constant upon reboot. We purchased a domain (b6foodtracker.com) for our web-app and redirected its traffic to the elastic IP of our web server. We also generated an SSL certificate from the free service Let’s Encrypt to enable HTTPS for better security.

The Django framework handles any incoming requests from the hardware components as well as displaying the inventory information to the user. As mentioned earlier, we use the default SQLite database to store user information. Django’s Model-View-Controller pattern provides us with a great interface to interact with the database on a high level, and we use model classes for the creation of different objects like users and devices. A class diagram can be found in the Appendix (Fig. 15).

When a user first visits the web-app, they are first greeted

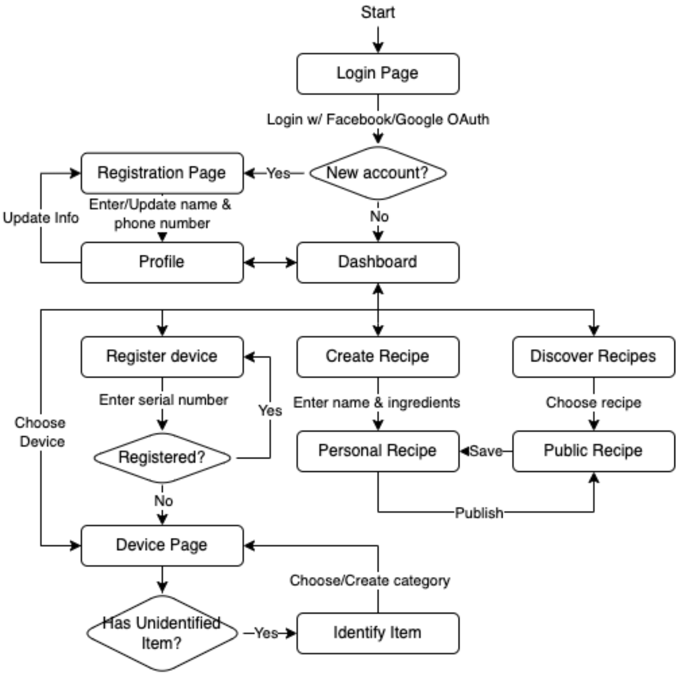


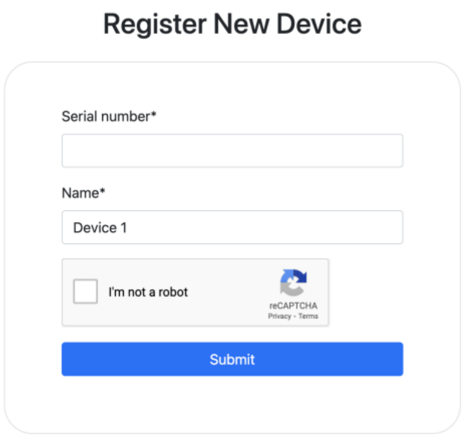
Fig. 8. Website flowchart.

with a splash page showcasing our product, and then prompted to register an account via OAuth with Google or Facebook. The registration attempt is only considered complete when the user enters a valid phone number—otherwise, we continuously redirect to the registration page. The phone number is needed in order to send shopping lists via SMS to the user.

User authentication is implemented using the Django component of the python-social-auth library called social-app-django, which supports logging in from a number of different services through the OAuth protocol, including Google, Facebook, Twitter, etc. We associate each user with their email address, so a user logging in from different providers with the same email address would log in to the same account in our system. The social-app-django library follows a login pipeline, which consists of functions that fetches the user's id from the provider, checks if the user already exists in the system, etc. We added custom functions to the pipeline to fetch users' profile pictures and email addresses. Fetching email addresses is needed because some OAuth providers (like Facebook) don't provide this information by default. To comply with OAuth requirements, we also have a privacy policy page stating the reason and scope of our minimal user data collection.

Phone number validation is implemented using the django-phonenumber-field library, which provides a phone number field for our object models. It can parse and validate phone numbers by checking the area code (first three digits), the central office code (middle three digits), and the line number (last four digits). Currently we only support US numbers, but this can be easily expanded in the future.

After the user is authenticated, the user must register a device with the web-app in order to use the service. We maintain a list of all the devices we have "manufactured" to prevent fraudulent device registration. When the user enters the serial number of their new device during device registration, we activate this device and assign the current user as its owner. To prevent malicious actors brute forcing all possible serial numbers, we added reCAPTCHA (a CAPTCHA system developed by Google) to the registration form to block any automated registrations. Relatedly, attempting to register an already-registered device either (1) shows a generic error message, or (2) shows a warning message if the device in question is already assigned to the current user. Showing a generic error message was a conscious choice to leak as little information to the user as possible to minimize potential attacks by a malicious actor.



Register New Device

Serial number*

Name*

Device 1

Fig. 9. Screenshot of the device registration page.

When handling incoming POST requests from the hardware module, the web-app invokes the CV component for object classification. If the CV component is able to classify the object, the web-app updates the inventory in the database accordingly. Otherwise, the web-app classifies the item as an unregistered item, and keeps a thumbnail of the object obtained from SSIM difference calculations in the CV component. When viewing any storage location, the web-app displays a list of unregistered items, and a prompt for users to manually classify them. If the user interacts with the prompt, the website displays the thumbnail image and prompts the user to select from the existing categories, or manually enter an item category.

Each identified item is assigned a location field to indicate the device that currently holds the item. These items are then filtered through a Queryset by location on the server side to generate an inventory list on the web-app. If the user owns multiple devices, the web-app can also display a combined inventory with a Queryset of all devices owned by a User, which is useful to see the combined ingredients currently in the kitchen, perhaps for any recipes that call for ingredients stored in multiple locations.

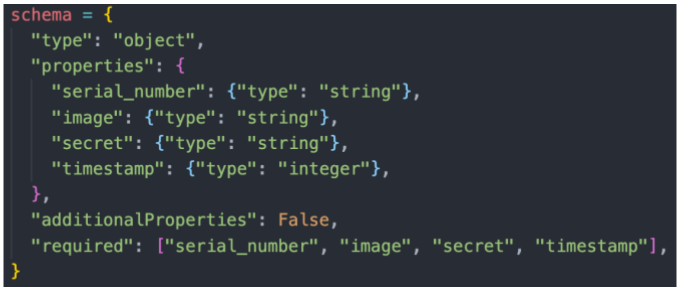
We designed two models for the recipes, Recipe and PublicRecipe. The former is private and only viewable by its creator while the latter is viewable by anyone. When a user creates a recipe, we use the Recipe model, and if the user decides to publish it, we copy its contents and create a new instance of the PublicRecipe with them. Similarly, we create a new Recipe instance if the user wants to save a public recipe. Each recipe contains a list of ingredient Categories, and we iterate through all items a user owns and check against this list to generate a shopping list upon request.

C. *Communications*

The hardware component and the cloud server communicate through HTTP POST requests. The POST requests are generated via python's requests package and its content is a serialized JSON string.

When the device takes a photo, it converts it to a string of utf characters using python's base64 package. (See Fig. 10 for full schema.)

The web-app only accepts requests coming from activated devices. It also stores the hashed secret key for each device in the database to validate requests. Using the secret key ensures



```
schema = {
  "type": "object",
  "properties": {
    "serial_number": {"type": "string"},
    "image": {"type": "string"},
    "secret": {"type": "string"},
    "timestamp": {"type": "integer"},
  },
  "additionalProperties": False,
  "required": ["serial_number", "image", "secret", "timestamp"],
}
```

Fig. 10. Schema for the JSON message.

that a malicious actor will not be able to modify someone else's inventory by simply modifying the serial number in the request; hashing the secret keys is an added security measure to ensure the system's operation in case of a data leak.

The hardware device also sends POST requests with an empty "image" JSON field every 15 minutes (we used 1 minute during the live demo in case we had to showcase the functionality) as heartbeat messages to the server to indicate its online status. If the server hasn't received any messages from an activated device for 45 minutes (missing three heartbeats), its owner is notified via an online/offline status icon on the website.

When the user visits the website, the client-side JavaScript code uses AJAX to pull information from the server every 10 seconds to get the most recent inventory. This step enables dynamic updates to the webpage when the inventory in the database is modified without the need to refresh the page.

D. CV Component

To review from Section V, a globally registered grocery item is a grocery item that can be recognized by all users' tracker. A locally registered grocery item is an item that can only be identified by a specific user's tracker.

At startup, the CV component extracts the features and key points from the iconic images for each of the 10 default globally registered image classes. The grocery items that are registered globally are as follows: Applesauce, crushed tomatoes can, shredded cheese bag, spaghetti, baking powder, yogurt, cereal, and Ritz crackers. The descriptors for each of these image classes are held in a global state variable, and are used until the web server shuts down.

When invoking the computer vision code, the web application passes the following information: image of previous state, image of new state, the set of items already present in the storage location, and a map of locally registered item categories to their descriptors. The CV component performs four passes in order to classify the object. A visual demonstration of each of these passes is provided in Fig. 11.

First, we undistort both of the images with two distortion coefficient matrices, K and D, which are obtained by taking pictures containing a 9x9 checkerboard and feeding them to an OpenCV algorithm that approximates the K and D values (cv2.fisheye.calibrate). We took 92 Checkerboard images in total. This process does not perfectly undistort the image; however, additional calibrations were providing only negligible improvements, and for our purposes the undistortion was working sufficiently well.

The CV component first performs a Structural Similarity Index Measure (SSIM) differentiation between the image of the previous state and the image of the new state. This is an algorithm that performs a pixel by pixel similarity measure, in such a way that is more in line with how humans perceive difference than a simple pixel diff [7]. Using SSIM helps us to mitigate the effects of slight changes in lighting. We then perform contour detection on the SSIM diff, to produce a list of regions of difference between the two images. We isolate the largest region of change, as we expect the added/removed item to be present in that region. This step occurs after the undistortion, as we have found it to work slightly better on undistorted images.

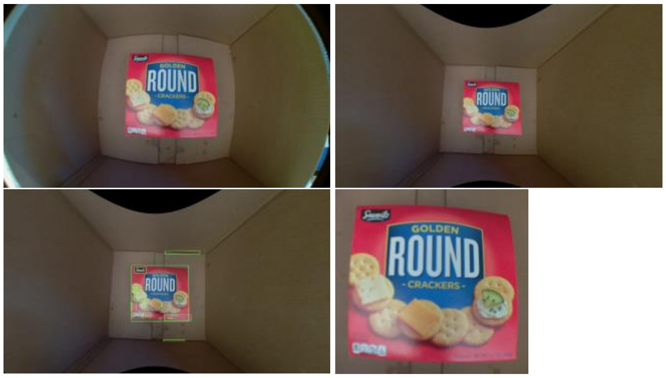


Fig. 11. Showcase of the CV processing. Initial raw image taken by RPI, top left. Undistorted image, top right. Image with annotated regions of change, lower left. Localized image, lower right.

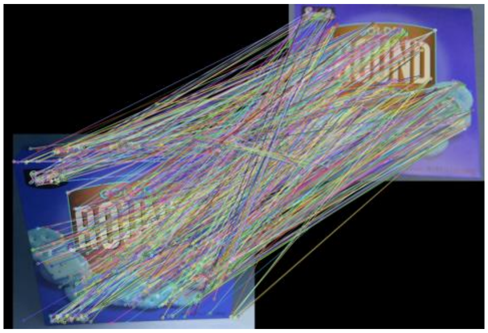


Fig. 12. Iconic image and target image, showing descriptors and keypoints of SIFT algorithm.

Next, we run SIFT to extract the keypoints and descriptors from the region of interest of the new state image. We then attempt to match the descriptors to each of the known grocery items' descriptors. If we fail to find any items of interest, we repeat the above process on the region of interest in the old state image, restricting the comparison to items that we know are present in that location.

Finally, we perform some heuristics to help mitigate common confusions. Notably, the largest region of change often captured large, and variable amounts of background. Therefore, most of these checks were designed with the assumption that the size of the region of change was often larger than the object, but very rarely smaller. This also meant that checking for the ratio of colors seen in an image was often inaccurate in the case that large swaths of background was present. The secondary checks included: Preemptively removing items from consideration if the region of change is smaller than the given item, deciding in favor of milk and Yogurt if the object was predominantly white, deciding in favor of crushed tomatoes if there was red in the image and SIFT was torn between tomatoes and applesauce. These were performed according to the most common confusions seen in our confusion matrix. Please see Tables V, VI, VII in the appendix for the confusion matrices obtained throughout testing, and the confusion matrix of the final product.

If we successfully identify any grocery items in the new state image, it means that the user has added that grocery item to the storage location. If we successfully identify any grocery items in the old state image, it means that the user has removed that item from the storage location. If we fail to identify any registered grocery items, it means that the user has added an unregistered item. The new item is given a temporary UNKNOWN_ITEM category by the web component, and its descriptors are then added to the user's map of registered grocery classes items. This is done so that if the item is removed prior to the user manually specifying the item, we can still identify it.

E. *Overall difference with our original design document*

Overall, there were few changes with respect to our design document. The largest changes include the addition of recipes, and the ability to share them on a public Discover page. There were also several additions to the CV process: the undistortion step, the secondary heuristic step, and the change from pure pixel diff to using SSIM.

There were also several changes in this section, where we added a bit of specificity that we hadn't decided on at the time or writing the design document: Apache, bootstrap CSS, etc.

VII. Test, Verification and Validation

In order to test our device, we repeatedly simulate updating a "kitchen" inventory. To do this, we subdivided our storage location into 9 general regions, all the cardinal/intercardinal regions (45°, 90°… 360°), and center. We then pick a random grocery item, and a random location in which to store that item. If the grocery item is already being stored, we first remove it. If the location where we are placing the item is occupied, we remove the items until we can fit our randomly chosen grocery item.

If the item could be rotated, it is placed in such a way as to best orient the label towards the camera. If the location is not center, we place the item as close to the wall of the container as we can. In the event that an error occurs, we manually update the information on the database so that it was accurate for the next update. We feel that this is a reasonable simulation of how a typical user might store an item in a storage location. This process was used for both Response time testing, and Classification accuracy testing. The procedures are listed in more detail below. (See the appendix Tables VIII and IX for the complete dataset.)

Originally, we wanted to have two distinct storage locations, and pick the location from between the two locations (IE, a random choice of 18 possible locations). However, we were unable to due to unfortunately breaking the CSI port on one of our RPi's, and being unable to procure a replacement in time (see Section VIII.E).

During testing, our website was hosted on t2.xlarge EC2 instance, which has 4 vCPUs and 16 GB memory.

A. *Response Time*

In order to test update speed (i.e. time from button press to website update), we simply measure the amount of time that passes from when the hardware component sends the item, to when the hardware component receives an acknowledgement. While this does not account for the travel time of the acknowledgement, that time should be fairly minor in the face of the server processing time (several seconds vs 10's of milliseconds) This provides us with a fairly accurate measure of the response time of a typical update to a user's storage location. This testing was done in conjunction with the classification accuracy test.

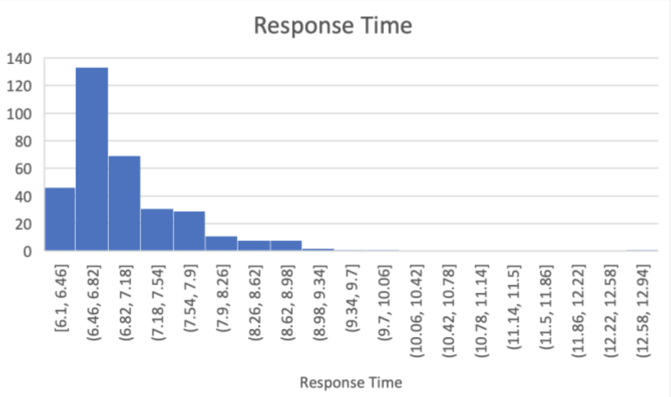


Fig. 13. Graph of response times.

Our results were very good: we had a mean response time of 7.006 seconds, a median response time of 6.78 seconds, and a worst case response time of 12.825 seconds. Therefore, we can confidently state that, in the general case, the response time is within 10 seconds. (See Fig. 13 for graph.)

B. *Classification Accuracy*

To check classification accuracy, we repeatedly updated the kitchen's inventory using the process described at the beginning of this section, keeping track of the success, failures, and misidentifications per item. (See Fig. 1 for how we classify different failures.) We repeat this until we've seen every item 10 times or we have 200 trials, whichever takes longer. Given that we expect the update process to closely simulate the behavior of a typical user, this should be a fairly accurate measure of classification accuracy for a typical user over the course of their using the device.

Our first round of testing took 272 trials (see Table VIII). We achieved an accuracy ratio of 93.75%, a false positive ratio of 6.25%, and a false negative ratio of 0%. This false positive ratio was greater than our allowed 5%, and we deemed it a failure.

We made some changes to the heuristic step, to make the heuristic functions more liberal in making changes. We also added a check for the difference between the most likely object class, and the second/third most likely class according to SIFT. If this difference wasn't sufficiently large, we would choose to fail to ID the item. We also tried putting blank white paper at the base of the storage location, to hopefully provide a more distinct background.

The overall result of the second round of testing was very poor (see Table IX). We called the testing early, at 58 trials. We achieved an accuracy ratio of 46.55%, false positive ratio of 5.17%, and a false negative ratio of 48.28%. We discontinued testing at this point, and reverted to the previous setup.

Overall, we failed to achieve our desired results, due to the overprevalence of false positives.

C. *Heartbeat Testing*

To test the heartbeat mechanism, we disconnected the RPI from the internet for 45 minutes, and then reconnected it. This test was successful. This testing was likely insufficient, as we did encounter one error at time of live demo, wherein the device would be set to offline when first registered, depending on the time that device was added to our database. This is now fixed.

VIII. Project Management

A. *Schedule*

We've divided our Gantt chart into subsections (Appendix, Fig. 16): CV proof of concept, Web App Component, Benchmarking, Integration, Website enhancements, and Documentation. The first three sections are the initial work that can be done in parallel, that are necessary for the MVP. Integration is self-explanatory. Website enhancements consist of a number of user quality of life improvements that, while important, are not needed for the MVP.

When comparing our original Gantt chart to our final version, there are several noticeable differences. First, work on CV optimizations and CSS continued pretty much up until the very end. This was caused by placing greater priority on other functionality, and since work on both CV optimization and CSS could be completed piecemeal, work was often start and stop. Second, AJAX took much longer than expected due to a relative lack of familiarity compared to other components of the project, which had a knock-on effect on delaying other components further downstream. Finally, some other smaller items that could be delayed until the final demo (hardware button, HTML/CSS improvements, etc.).

B. *Team Member Responsibilities*

Generally speaking, Keaton was primarily responsible for writing and testing the CV used for object classification. Harry also contributed some optimizations, specifically with regards to the SSIM localization. Harry was primarily responsible for the overall system design, webpage styling using Bootstrap, and web-app backend development including OAuth and backend logic. Jay helped with this, especially as it pertained to his primary work with the front end. Jay was primarily responsible for the web application front end (AJAX and HTML) and assisting implement the backend. Many responsibilities for the web-app switched hands between us three members as we found out which of us had more free time between course loads as well as maneuvering personal experience.

C. *Bill of Materials and Budget*

Overall, we are well within the bounds of our budget, sitting at $370 spent (see Table III).

Of our initial purchases at the beginning of the semester, there were several things we ended up not using: two of the three OV5647 camera modules, which may or may not be broken, and the lenses for those camera modules. Additionally, the two RPi's which we ordered did not actually arrive due to P-card ordering issues. We do not know for certain the final status of this order. Upon last talking with Quinn, the order had been canceled, but independent of our Capstone he had placed a separate order to have a spare RPi on hand. Even with both RPi's still included in our cost breakdown as a worst case scenario, we remained under budget.

D. *AWS Usage*

At the time of writing, the usage for our AWS credits is as follows:

TABLE II. AWS Cost breakdown

Service	***Hourly rate***	***Time (hrs)***	***Cost***
EC2 On Demand Linux t2.xlarge Instance	$0.1856	5.709	$1.06
EC2 On Demand Linux t3.2xlarge Instance	$0.3328	30	$9.98
Elastic IP addresses	$0.005	114.8	$0.57
Total cost:			***$11.61***

Fig. 14. Breakdown of AWS cost, per item, per hour.

Our AWS credits were used to run one EC2 instance which did dual duty running our CV code, which identified grocery items from a RPi camera module, and a web-app which was used to send that image from the camera to our CV code and display it on a web application in a user-friendly way. We launched a t2.xlarge instance for development and upgraded to a more powerful t3.2xlarge instance for our final capstone demo, as we had leftover credit. We also used the Elastic IP to attach to our purchased domain, b6foodtracker.com. In total, $11.61 of our allotted $50 were consumed at the time of writing. We'd like to express our gratitude to AWS for providing the Electrical and Computer Engineering department at Carnegie Mellon University with these free credits.

E. *Risk Management*

Most aspects of our project were done independently, and there was little to no risk of catastrophic failure. In the design document, we viewed the two most risky portions of the project to be the possibility of not meeting the accuracy requirements for the CV device, and not meeting the update speed requirements.

For the accuracy requirements, we did end up adding several heuristic checks, and performing many tweaks. Unfortunately, we were unsuccessful in our ability to meet the accuracy requirements. The major timesink actually ended up being the time required to perform the test, it took about three hours to take all the photos for the first test. A more thorough test should have been performed earlier into the project, or we should have come up with a faster way to perform a unit test on the CV component itself.

The update speed requirements actually ended up being a non issue, we were well within the bounds without the need for any optimizations. Additionally, as we mentioned in the design report, we were able to even further speed up the overall response time by using a higher capacity EC2 instance.

Besides the two risks we identified going into the project, we also had an unexpected hardware issue when the CSI port on our RPi broke, and we didn't have a replacement. Additionally, our order for a second RPI that we had ordered to demonstrate multiple concurrent users was canceled due to logistical difficulties with the CMU P-card ordering procedures. Thankfully, Quinn was very helpful, and loaned us a replacement that he had on hand that also worked with our hardware specifications mentioned in Section VI.A, *Hardware System Implementation*.

IX. Ethical Issues

First, given that we can only guarantee accuracy for the set of items we've explicitly supported, there is the potential unfairness to users who generally use grocery items different than those that we explicitly support. This may be due to individual variance, or perhaps users that belong to a culture that generally uses different grocery items. Additionally different cultures may use different labeling techniques, which may have effects on the accuracy of SIFT. For example, labels containing Japanese hiragana may be more/less recognizable by SIFT, which would lead to increased/decreased accuracy. While this certainly could become an issue, it is difficult to diagnose, given the vast number of different written languages, and grocery items. We view this as an issue to diagnose once if/when we decide to expand this product to an international marketplace.

Second, was the potential for using our device as a spycam. IE, a user could place the device in a public location, hotwire the hardware unit to take a photo every few seconds, and scrape the relevant information from the web application. Due to the nature of our design, it would need to be placed in a location with consistent lighting, and whatever object/entity it would be looking for must be easily recognizable by the SIFT. This would only be doable by a very knowledgeable malicious user, and it would likely be easier and much more effective for such a user to build their own spycam rather than retrofit our project.

Third, systems that require registration can be exploited by malicious actors brute forcing all the possible inputs, which in our case is the serial number of the devices. Since we have incorporated OAuth in our system for secure authentication, and implemented secret key hashing and reCAPTCHA as safety measures, our system is tolerant against such attacks.

Fourth, since we store private information about users like email and phone number, data leaks could have serious consequences. Therefore, the security of the system needs to be upgraded and rigorously tested in the future. Additionally, we have a privacy policy that clearly delineates the scope and intended use of any personal information, and any opt-out procedures for deleting a user's account.

Lastly, since our system supports sharing recipes, which consists of texts like recipe names given by the user, there is a possibility for malicious users to spread illicit or hateful words. However, this is easily solvable in the future by setting up a monitoring system.

X. Related Work

There were also a number of similar projects from previous years that we investigated while working on designing our own. "Backpack buddy", Spring 2021 C0, is a project that tracks the location of the user's items using RFID tags. It was our principal source of inspiration before we switched to using computer vision. "SmolKat", Spring 2021 D3, is another similar project that focused on identifying items in a storage location. They used Google's Cloud Vision API as their classification system, as opposed to SIFT. "Sous-Chef" Spring 2020 A4, also focused on identifying items in a storage location using CV. However, they focused on looking for barcodes, as a method of identifying the item. "Fresh Eyes," from this Spring 2022 B3, also developed a kitchen inventory app, but their use case focused more on produce expiration dates. As mentioned in an earlier section, we came across a similar project online, "Grocer Eye" which we used as a reference for our potential accuracy.

XI. Summary

Overall, while we were unable to meet all of our design requirements, we came very close. Therefore, we feel fairly happy with the state of our project.

A. *Future work*

Given more time, the first priority would be to optimize the CV component, such that we could meet our design requirements. I think this could be done fairly easily, given how close we were.

Supporting a greater list of items, and possibly using a CNN could easily be managed with more time, and with more people available using the product to generate training data for the CV algorithm.

Several stretch goals for the website, such as integrating some online recipes functionality, would have been a fantastic addition to our system had we had more time. User-created recipes supporting notes for the actual recipes would also have made our product more holistic.

Given a huge amount of extra time and funding, we could try and create an actual demo smart appliance, as opposed to just using a cardboard box. Perhaps the increased stability from such a hardware build would also improve pixel diff calculations in our CV component.

B. *Lessons Learned*

Personally, we feel that there were two major lessons. First, when working with embedded devices, cost permitting, have backup hardware in case of emergency. Secondly, in situations where performance is paramount, do rigorous unit testing as early as possible, and ensure that your unit testing can be performed fast enough that you can iterate on the results as needed.

C. *Closing Thoughts*

Our inspiration for this project has deep personal relevance to us all. We feel that this problem of grocery inventory is overwhelmingly common, and we hope that someday our project can be improved upon, and made into a commercially successful tracker that will fulfill the grocery-shopping needs of every home cook. Thank you for reading.

GLOSSARY OF ACRONYMS

AWS - Amazon Web Services

BRIEF - Binary Robust Independent Elementary Features

CAPTCHA - Completely Automated Public Turing test to tell Computers and Humans Apart

CSI - Camera Serial Interface

CV - Computer Vision

EC2 - Amazon Elastic Cloud Compute

GPIO - General-Purpose Input/Output

HTTP - Hypertext Transfer Protocol

JSON - JavaScript Object Notation

MVP - Minimum Viable Product

OAuth - Open Authorization

ORB - Oriented FAST and Rotated BRIEF

RPi – Raspberry Pi

SIFT - Scale Invariant Feature Transform

SMTP - Simple Mail Transfer Protocol

SSIM - Structural Similarity Index Measurement

SSL - Secure Sockets Layer

YOLO - You Only Look Once

REFERENCES

[1] "How to Check, Measure, and Improve Server and Application Response Time With Monitoring Tools," DNSstuff, 12-Dec-2019. Accessed on Feb 28, 2022 [Online]. Available: https://www.dnsstuff.com/response-time-monitoring.

[2] R. M. Bhimani, "Grocereye - a YOLO model for grocery object detection," Rehaan M. Bhimani, 18-Dec-2020. Accessed on Mar 3, 2022 [Online]. Available: http://students.washington.edu/bhimar/highlights/2020-12-18-GrocerEye/.

[3] D. Lowe, “Object Recognition from Local Scale-Invariant Features.” [Online]. Available: https://www.cs.ubc.ca/~lowe/papers/iccv99.pdf

[4] K. Avis-Riordan, “It’s official: we’re a nation of fridge raiders,” House Beautiful, Jul. 28, 2017. https://www.housebeautiful.com/uk/lifestyle/storage/news/a2110/fridge-food-cupboard-habits/ (accessed May 08, 2022).

[5] M. King, “7 Best Restaurant Inventory Management Software for 2022,” Fit Small Buisness, Nov. 2021. https://fitsmallbusiness.com/restaurant-inventory-management-software/ (accessed May 07, 2022).

[6] “SuperCook: Recipes By Ingredient - Apps on Google Play,” play.google.com. https://play.google.com/store/apps/details?id=com.supercook.app

[7] Zhou Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," in IEEE Transactions on Image Processing, vol. 13, no. 4, pp. 600-612, April 2004, doi: 10.1109/TIP.2003.819861.

[8] D. Lowe, “Distinctive Image Features from Scale-Invariant Keypoints,” Jan. 2004. Accessed: May 07, 2022. [Online]. Available: https://people.eecs.berkeley.edu/~malik/cs294/lowe-ijcv04.pdf

APPENDIX

TABLE III. BILL OF MATERIALS

Description	**Model #**	**Manufacturer**	**Quantity**	**Cost @**	**Total**
RPi Model B	4328498196	RPi Foundation	2*	$100	$200
1/2.7"mm Focal Length Lens		Arducam	2	$18	$36
OV5647 with M12 Lens Preattached		Arducam	3	$28	$84
AWS Credit		Amazon	1	$50	$50
*Order may or may not have gone through					
				Grand Total	**$370.00**

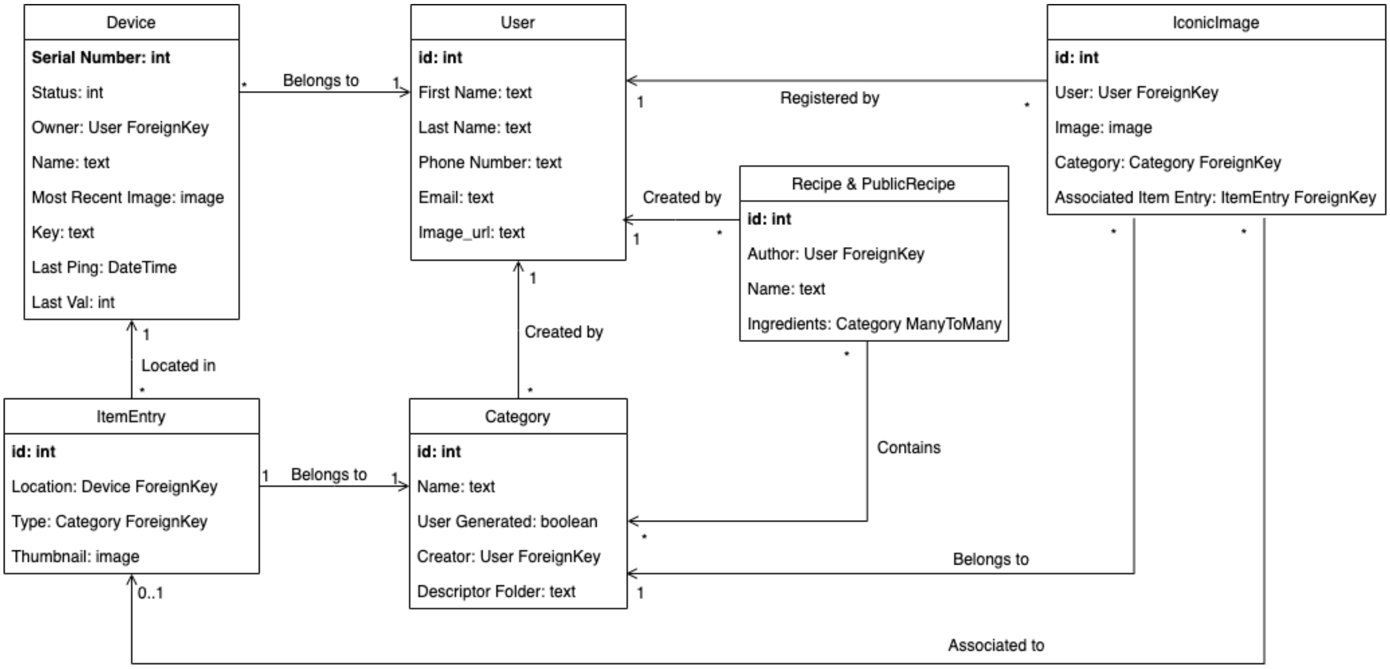
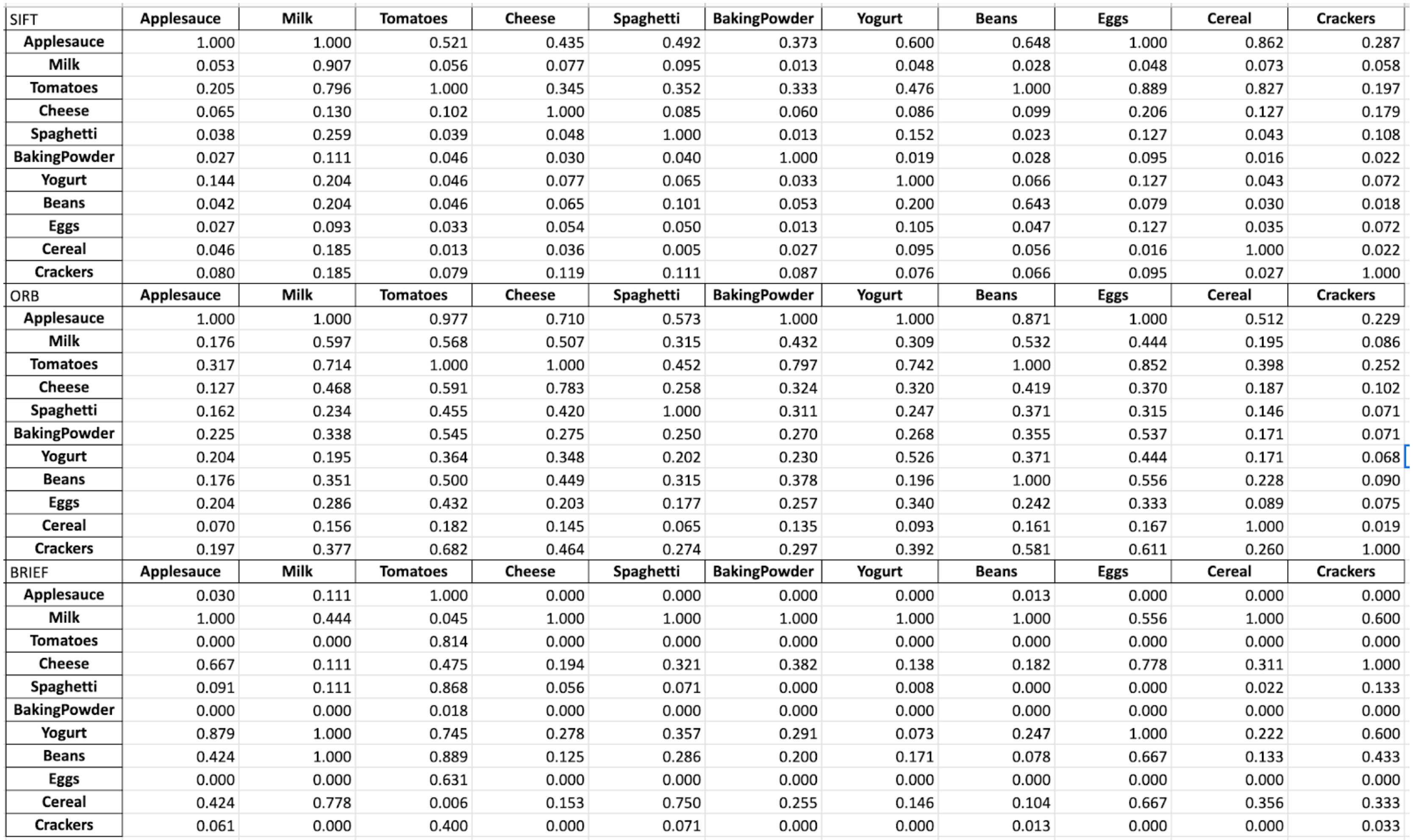


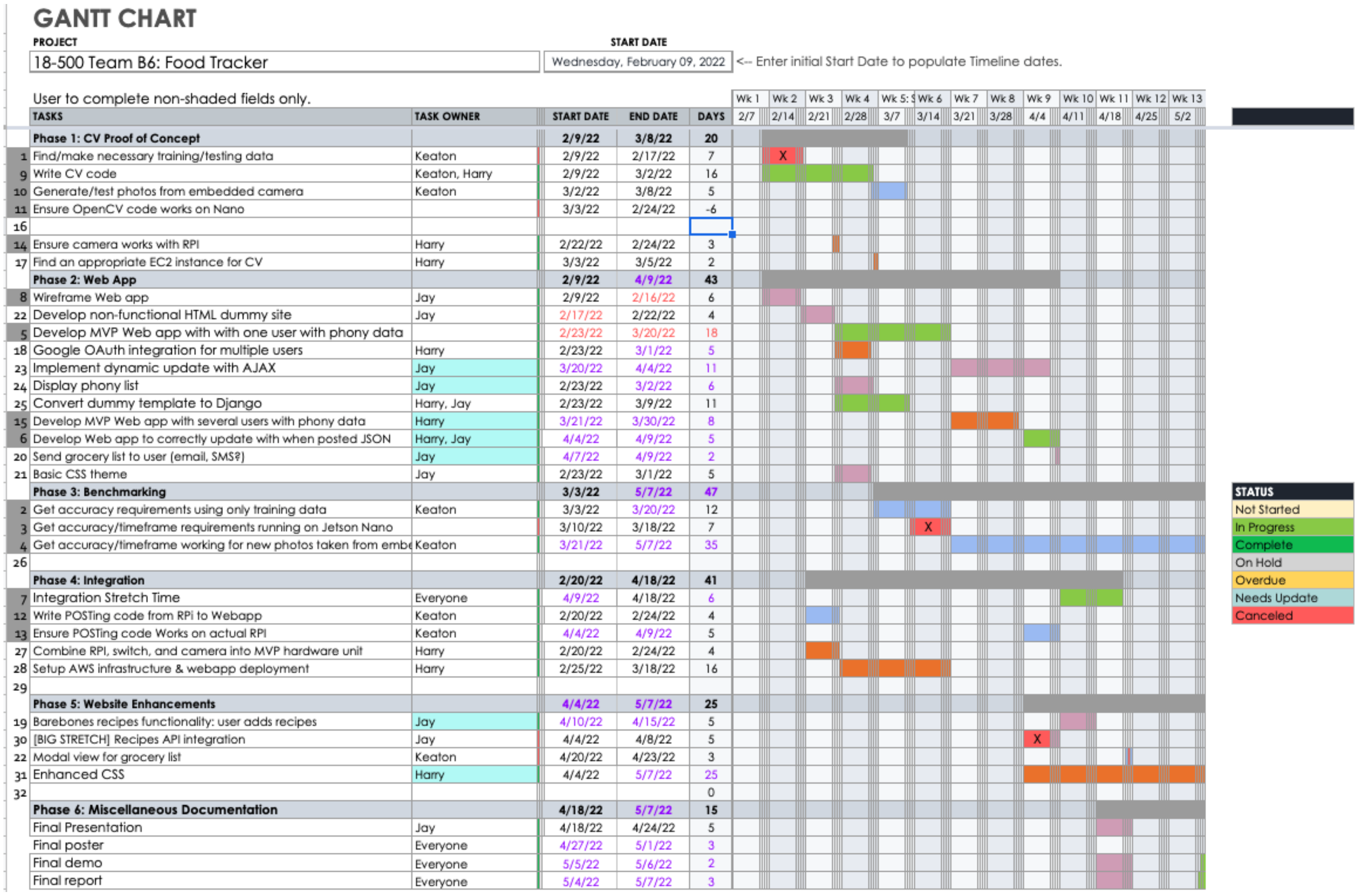
Fig. 15. Class model diagram of database



SIFT	Applesauce	Milk	Tomatoes	Cheese	Spaghetti	BakingPowder	Yogurt	Beans	Eggs	Cereal	Crackers
Applesauce	1.000	1.000	0.521	0.435	0.492	0.373	0.600	0.648	1.000	0.862	0.287
Milk	0.053	0.907	0.056	0.077	0.095	0.013	0.048	0.028	0.048	0.073	0.058
Tomatoes	0.205	0.796	1.000	0.345	0.352	0.333	0.476	1.000	0.889	0.827	0.197
Cheese	0.065	0.130	0.102	1.000	0.085	0.060	0.086	0.099	0.206	0.127	0.179
Spaghetti	0.038	0.259	0.039	0.048	1.000	0.013	0.152	0.023	0.127	0.043	0.108
BakingPowder	0.027	0.111	0.046	0.030	0.040	1.000	0.019	0.028	0.095	0.016	0.022
Yogurt	0.144	0.204	0.046	0.077	0.065	0.033	1.000	0.066	0.127	0.043	0.072
Beans	0.042	0.204	0.046	0.065	0.101	0.053	0.200	0.643	0.079	0.030	0.018
Eggs	0.027	0.093	0.033	0.054	0.050	0.013	0.105	0.047	0.127	0.035	0.072
Cereal	0.046	0.185	0.013	0.036	0.005	0.027	0.095	0.056	0.016	1.000	0.022
Crackers	0.080	0.185	0.079	0.119	0.111	0.087	0.076	0.066	0.095	0.027	1.000
ORB	**Applesauce**	**Milk**	**Tomatoes**	**Cheese**	**Spaghetti**	**BakingPowder**	**Yogurt**	**Beans**	**Eggs**	**Cereal**	**Crackers**
Applesauce	1.000	1.000	0.977	0.710	0.573	1.000	1.000	0.871	1.000	0.512	0.229
Milk	0.176	0.597	0.568	0.507	0.315	0.432	0.309	0.532	0.444	0.195	0.086
Tomatoes	0.317	0.714	1.000	1.000	0.452	0.797	0.742	1.000	0.852	0.398	0.252
Cheese	0.127	0.468	0.591	0.783	0.258	0.324	0.320	0.419	0.370	0.187	0.102
Spaghetti	0.162	0.234	0.455	0.420	1.000	0.311	0.247	0.371	0.315	0.146	0.071
BakingPowder	0.225	0.338	0.545	0.275	0.250	0.270	0.268	0.355	0.537	0.171	0.071
Yogurt	0.204	0.195	0.364	0.348	0.202	0.230	0.526	0.371	0.444	0.171	0.068
Beans	0.176	0.351	0.500	0.449	0.315	0.378	0.196	1.000	0.556	0.228	0.090
Eggs	0.204	0.286	0.432	0.203	0.177	0.257	0.340	0.242	0.333	0.089	0.075
Cereal	0.070	0.156	0.182	0.145	0.065	0.135	0.093	0.161	0.167	1.000	0.019
Crackers	0.197	0.377	0.682	0.464	0.274	0.297	0.392	0.581	0.611	0.260	1.000
BRIEF	**Applesauce**	**Milk**	**Tomatoes**	**Cheese**	**Spaghetti**	**BakingPowder**	**Yogurt**	**Beans**	**Eggs**	**Cereal**	**Crackers**
Applesauce	0.030	0.111	1.000	0.000	0.000	0.000	0.000	0.013	0.000	0.000	0.000
Milk	1.000	0.444	0.045	1.000	1.000	1.000	1.000	1.000	0.556	1.000	0.600
Tomatoes	0.000	0.000	0.814	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cheese	0.667	0.111	0.475	0.194	0.321	0.382	0.138	0.182	0.778	0.311	1.000
Spaghetti	0.091	0.111	0.868	0.056	0.071	0.000	0.008	0.000	0.000	0.022	0.133
BakingPowder	0.000	0.000	0.018	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Yogurt	0.879	1.000	0.745	0.278	0.357	0.291	0.073	0.247	1.000	0.222	0.600
Beans	0.424	1.000	0.889	0.125	0.286	0.200	0.171	0.078	0.667	0.133	0.433
Eggs	0.000	0.000	0.631	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cereal	0.424	0.778	0.006	0.153	0.750	0.255	0.146	0.104	0.667	0.356	0.333
Crackers	0.061	0.000	0.400	0.000	0.071	0.000	0.000	0.013	0.000	0.000	0.033

TABLE IV. COMPARISON OF VARIOUS CV ALGORITHMS.

The x-axis is the iconic image, and the y-axis is the actual item. 1.000 represents the best number of descriptor matches for the given iconic class. .5 would represent .5 * the best number of descriptor matches for the given iconic class.



GANTT CHART

PROJECT: 18-500 Team B6: Food Tracker

START DATE: Wednesday, February 09, 2022 <-- Enter initial Start Date to populate Timeline dates.

User to complete non-shaded fields only.

Wk 1 2/7 | Wk 2 2/14 | Wk 3 2/21 | Wk 4 2/28 | Wk 5: S 3/7 | Wk 6 3/14 | Wk 7 3/21 | Wk 8 3/28 | Wk 9 4/4 | Wk 10 4/11 | Wk 11 4/18 | Wk 12 4/25 | Wk 13 5/2

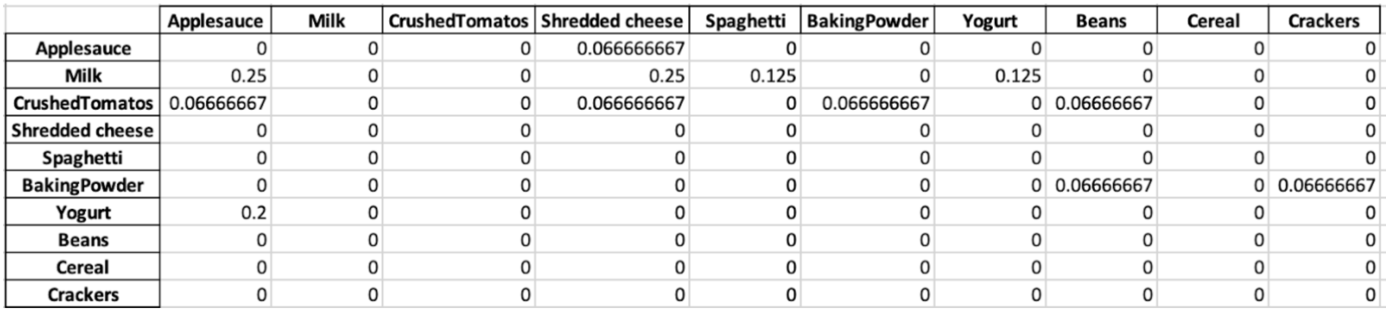
	TASKS	TASK OWNER	START DATE	END DATE	DAYS
	Phase 1: CV Proof of Concept		**2/9/22**	**3/8/22**	**20**
1	Find/make necessary training/testing data	Keaton	2/9/22	2/17/22	7
9	Write CV code	Keaton, Harry	2/9/22	3/2/22	16
10	Generate/test photos from embedded camera	Keaton	3/2/22	3/8/22	5
11	Ensure OpenCV code works on Nano		3/3/22	2/24/22	-6
16					
14	Ensure camera works with RPI	Harry	2/22/22	2/24/22	3
17	Find an appropriate EC2 instance for CV	Harry	3/3/22	3/5/22	2
	Phase 2: Web App		**2/9/22**	**4/9/22**	**43**
8	Wireframe Web app	Jay	2/9/22	2/16/22	6
22	Develop non-functional HTML dummy site	Jay	2/17/22	2/22/22	4
5	Develop MVP Web app with with one user with phony data		2/23/22	3/20/22	18
18	Google OAuth integration for multiple users	Harry	2/23/22	3/1/22	5
23	Implement dynamic update with AJAX	Jay	3/20/22	4/4/22	11
24	Display phony list	Jay	2/23/22	3/2/22	6
25	Convert dummy template to Django	Harry, Jay	2/23/22	3/9/22	11
15	Develop MVP Web app with several users with phony data	Harry	3/21/22	3/30/22	8
6	Develop Web app to correctly update with when posted JSON	Harry, Jay	4/4/22	4/9/22	5
20	Send grocery list to user (email, SMS?)	Jay	4/7/22	4/9/22	2
21	Basic CSS theme	Jay	2/23/22	3/1/22	5
	Phase 3: Benchmarking		**3/3/22**	**5/7/22**	**47**
2	Get accuracy requirements using only training data	Keaton	3/3/22	3/20/22	12
3	Get accuracy/timeframe requirements running on Jetson Nano		3/10/22	3/18/22	7
4	Get accuracy/timeframe working for new photos taken from embe	Keaton	3/21/22	5/7/22	35
26					
	Phase 4: Integration		**2/20/22**	**4/18/22**	**41**
7	Integration Stretch Time	Everyone	4/9/22	4/18/22	6
12	Write POSTing code from RPi to Webapp	Keaton	2/20/22	2/24/22	4
13	Ensure POSTing code Works on actual RPI	Keaton	4/4/22	4/9/22	5
27	Combine RPI, switch, and camera into MVP hardware unit	Harry	2/20/22	2/24/22	4
28	Setup AWS infrastructure & webapp deployment	Harry	2/25/22	3/18/22	16
29					
	Phase 5: Website Enhancements		**4/4/22**	**5/7/22**	**25**
19	Barebones recipes functionality: user adds recipes	Jay	4/10/22	4/15/22	5
30	[BIG STRETCH] Recipes API integration	Jay	4/4/22	4/8/22	5
22	Modal view for grocery list	Keaton	4/20/22	4/23/22	3
31	Enhanced CSS	Harry	4/4/22	5/7/22	25
32					0
	Phase 6: Miscellaneous Documentation		**4/18/22**	**5/7/22**	**15**
	Final Presentation	Jay	4/18/22	4/24/22	5
	Final poster	Everyone	4/27/22	5/1/22	3
	Final demo	Everyone	5/5/22	5/6/22	2
	Final report	Everyone	5/4/22	5/7/22	3

STATUS
Not Started
In Progress
Complete
On Hold
Overdue
Needs Update
Canceled

Fig. 16. Gantt chart.

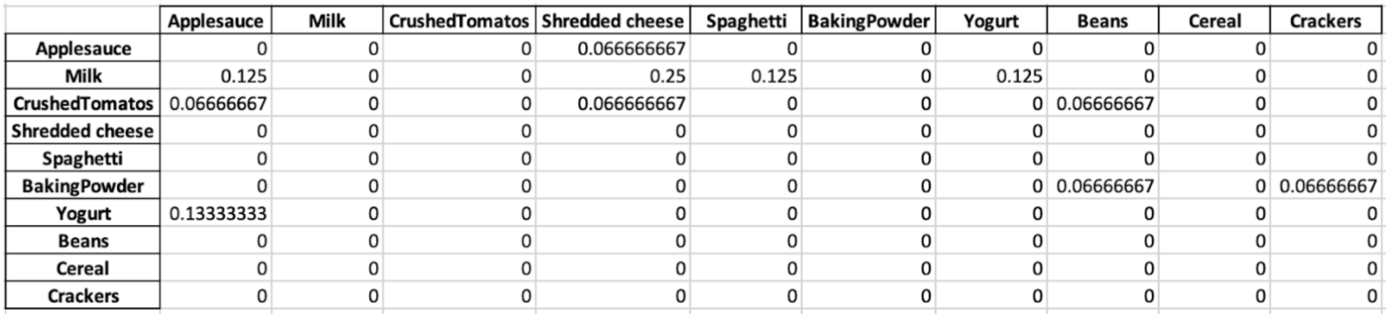
Note, for tables V and VI, orientable items (Applesauce, Yogurt, Beans, Crushed Tomato Can, etc.) were not necessarily placed angled towards the camera. As such, **confusions may be overrepresented** for those objects compared to the table VII, where printable objects were always oriented towards the camera.

TABLE V. CONFUSION MATRIX OF TESTING DATA LOCATED IN THE "TOPDOWN" FOLDER (~100 IMAGES), PRIOR TO IMPLEMENTATION OF HEURISTICS.



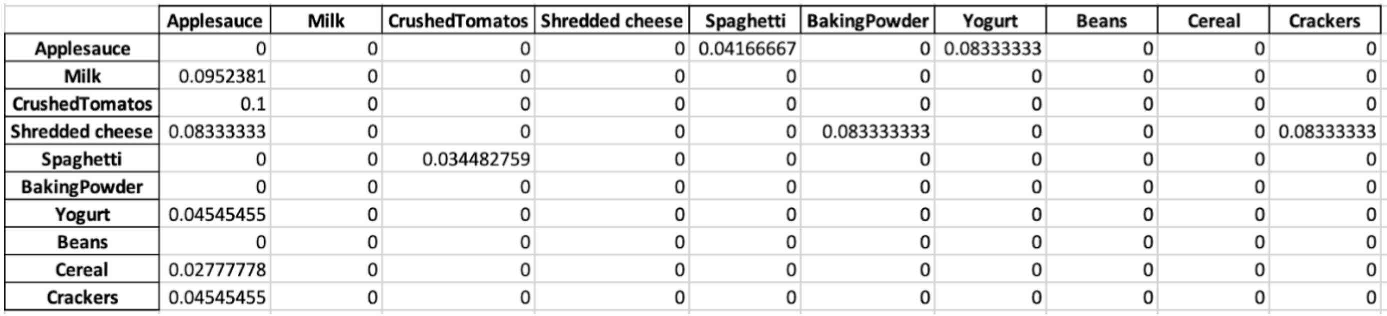
	Applesauce	Milk	CrushedTomatos	Shredded cheese	Spaghetti	BakingPowder	Yogurt	Beans	Cereal	Crackers
Applesauce	0	0	0	0.066666667	0	0	0	0	0	0
Milk	0.25	0	0	0.25	0.125	0	0.125	0	0	0
CrushedTomatos	0.06666667	0	0	0.066666667	0	0.066666667	0	0.06666667	0	0
Shredded cheese	0	0	0	0	0	0	0	0	0	0
Spaghetti	0	0	0	0	0	0	0	0	0	0
BakingPowder	0	0	0	0	0	0	0	0.06666667	0	0.06666667
Yogurt	0.2	0	0	0	0	0	0	0	0	0
Beans	0	0	0	0	0	0	0	0	0	0
Cereal	0	0	0	0	0	0	0	0	0	0
Crackers	0	0	0	0	0	0	0	0	0	0

TABLE VI. CONFUSION MATRIX OF TESTING DATA LOCATED IN THE "TOPDOWN" FOLDER (~100 IMAGES), POST IMPLEMENTATION OF HEURISTICS.



	Applesauce	Milk	CrushedTomatos	Shredded cheese	Spaghetti	BakingPowder	Yogurt	Beans	Cereal	Crackers
Applesauce	0	0	0	0.066666667	0	0	0	0	0	0
Milk	0.125	0	0	0.25	0.125	0	0.125	0	0	0
CrushedTomatos	0.06666667	0	0	0.066666667	0	0	0	0.06666667	0	0
Shredded cheese	0	0	0	0	0	0	0	0	0	0
Spaghetti	0	0	0	0	0	0	0	0	0	0
BakingPowder	0	0	0	0	0	0	0	0.06666667	0	0.06666667
Yogurt	0.13333333	0	0	0	0	0	0	0	0	0
Beans	0	0	0	0	0	0	0	0	0	0
Cereal	0	0	0	0	0	0	0	0	0	0
Crackers	0	0	0	0	0	0	0	0	0	0

TABLE VII. CONFUSION MATRIX OF FINAL TRIAL (~270 IMAGES)



	Applesauce	Milk	CrushedTomatos	Shredded cheese	Spaghetti	BakingPowder	Yogurt	Beans	Cereal	Crackers
Applesauce	0	0	0	0	0.04166667	0	0.08333333	0	0	0
Milk	0.0952381	0	0	0	0	0	0	0	0	0
CrushedTomatos	0.1	0	0	0	0	0	0	0	0	0
Shredded cheese	0.08333333	0	0	0	0	0.083333333	0	0	0	0.08333333
Spaghetti	0	0	0.034482759	0	0	0	0	0	0	0
BakingPowder	0	0	0	0	0	0	0	0	0	0
Yogurt	0.04545455	0	0	0	0	0	0	0	0	0
Beans	0	0	0	0	0	0	0	0	0	0
Cereal	0.02777778	0	0	0	0	0	0	0	0	0
Crackers	0.04545455	0	0	0	0	0	0	0	0	0

TABLE VIII. FINAL TRIAL (~270 IMAGES)

Item	***Action***	***Location***	***Failed***	***Time***
cereal	add		0	7.74
cereal	remove		0	8.64
beans	add		0	8.593
Tomato Can	add		0	7.196
beans	remove		0	6.651
beans	add		0	6.932
beans	remove		0	7.392
beans	add		0	6.91
baking powder	add		0	6.98
cracker	add		0	7.296
baking powder	remove		0	7.67
baking powder	add		0	8.02
tomato Can	remove		0	8.63
cracker	remove		0	8.635
cracker	add		0	7.51
baking powder	remove		0	6.99
cracker	remove		0	7.555
baking powder	add		0	6.84
baking powder	remove		0	6.42
cereal	add		0	7.607
spaghetti	add		0	7.583
spaghetti	remove		0	6.913
cereal	remove		0	7.036
milk	add		0	8.647
cheese	add		0	7.06
beans	remove		0	6.84
cheese	remove		0	8.44
applesauce	add		0	7.48
tomato Can	add		0	6.71
tomato Can	remove		0	7.27
milk	remove		0	9.19
cereal	add		0	7.8658
cereal	remove		0	7.64
cracker	add		0	7.917
cracker	remove		0	7.194
baking powder	add		0	7.4
baking powder	remove		0	6.615
cheese	add		1	6.74
cheese	remove		1	7.151
applesauce	remove		0	8.42
applesauce	add		0	7.39
beans	add		0	7.79
baking powder	add		0	6.65
yogurt	add		0	6.51
baking powder	remove		0	6.59
spaghetti	add		0	6.63
baking powder	add		0	6.59
spaghetti	remove		0	6.8
cereal	add		1	8.21
kidney beans	remove		0	6.62
applesauce	remove		0	6.92
spaghetti	add		0	6.66
spaghetti	remove		0	7.48
spaghetti	add		0	6.86
spaghetti	remove		0	9.12
cereal	remove		0	7.77
spaghetti	add		0	6.67
beans	add	top	0	6.65
yogurt	remove	left	0	8.03
baking powder	remove	bl	0	8.9
baking powder	add	left	0	8.04
tomato Can	add	br	0	7.59
beans	remove	top	0	6.75
cracker	add	top	0	8.4
spaghetti	remove	right	0	7.78
cracker	remove	top	0	8.02
spaghetti	add	center	0	7.78
baking powder	remove	left	0	8.13
cereal	add	tl	0	7.86
cereal	remove	tl	0	8.08
cracker	add	tl	0	7.65
cracker	remove	tl	0	8.25
spaghetti	remove	center	1	8.35
cracker	add	b	0	8.72
tomato Can	remove	br	0	7.71
cheese	add	br	1	7.32
cheese	remove	br	1	6.81
milk	add	right	0	7.43
cracker	remove	b	0	7.12
cereal	add	b	0	7.86
beans	add	tl	0	6.55
milk	remove	right	0	8.48
yogurt	add	br	0	6.86
cracker	add	left	0	7.17
cracker	remove	left	0	6.91
tomato Can	add	left	0	7.1
spaghetti	add	tr	0	7.24
beans	remove	tl	0	6.48
beans	add	top	0	6.84
beans	remove	top	0	6.78
cereal	remove	b	0	7.12
beans	add	center	0	6.97
beans	remove	center	0	6.74
spaghetti	remove	tr	0	7.46
spaghetti	add	center	0	6.73
tomato Can	remove	left	0	7.12
cracker	add	left	0	7.58
spaghetti	remove	center	0	6.59
cracker	remove	left	0	7.52
spaghetti	add	left	0	7.49
applesauce	add	b	0	7.03
applesauce	remove	b	0	6.93
applesauce	add	tr	0	6.86
yogurt	remove	br	0	6.65
applesauce	remove	tr	1	6.84
cracker	add	right	1	7.51
yogurt	add	center	0	6.87
cracker	remove	right	1	6.9
beans	add	right	0	6.93
applesauce	add	b	0	6.94
yogurt	remove	center	0	7.83
milk	add	center	0	7.83
applesauce	remove	b	0	7.55
beans	remove	right	0	6.5
cracker	add	tr	0	6.599
cracker	remove	tr	0	6.448
cereal	add	br	0	7.09
yogurt	add	left	0	6.57
cereal	remove	br	0	6.54
beans	add	b	0	6.61
yogurt	remove	left	0	6.47
milk	add	tl	0	6.635
yogurt	add	bl	0	6.78
milk	remove	tl	0	6.9
beans	remove	b	0	6.882
cereal	add	top	0	7.18
cereal	remove	top	0	6.86
applesauce	add	bl	0	8.85
applesauce	remove	bl	0	6.83
spaghetti	add	tl	0	6.39
yogurt	remove	br	0	6.38
cereal	add	right	0	6.83
spaghetti	remove	tr	0	6.58
yogurt	add	Left	0	7.09
cereal	remove	right	0	9.993
applesauce	add	center	1	7.28
applesauce	remove	center	1	6.626

milk	add	right	0	6.52
milk	remove	right	0	7.59
yogurt	remove	Left	0	6.57
cereal	add	br	0	6.87
baking powder	add	tl	0	6.297
cereal	remove	br	0	7.79
cheese	add	tr	0	6.87
cheese	remove	tr	0	6.56
cheese	add	tr	0	6.5
applesauce	add	Left	0	6.66
cheese	remove	tr	0	6.37
cheese	add	right	0	6.9
cheese	remove	right	0	6.29
beans	add	tr	0	6.66
yogurt	add	b	0	6.76
yogurt	remove	b	0	7.89
cheese	add	br	0	6.75
applesauce	remove	Left	0	8.18
cracker	add	bl	0	6.725
cracker	remove	bl	0	6.52
tomato Can	add	center	0	6.69
tomato Can	remove	center	0	6.56
milk	add	bl	1	7.22
cheese	remove	br	0	6.47
tomato Can	add	br	0	6.63
yogurt	add	center	0	6.54
beans	remove	tr	0	6.5
beans	add	tr	0	6.5
baking powder	remove	tl	0	6.57
milk	remove	bl	1	7.67
baking powder	add	bl	0	6.39
yogurt	remove	center	0	6.45
cracker	add	tl	0	6.67
cracker	remove	tl	0	6.69
spaghetti	add	t	0	6.75
tomato Can	remove	br	0	6.657
beans	remove	tr	0	6.45
tomato Can	add	tr	1	6.3
spaghetti	remove	t	0	6.89
cheese	add	b	0	6.78
baking powder	remove	bl	0	8.75
baking powder	add	t	0	6.78
cheese	remove	br	0	6.39
tomato Can	remove	tr	0	6.3
milk	add	right	0	6.919
baking powder	remove	t	0	6.82
baking powder	add	center	0	12.825
milk	remove	right	0	7.65
cheese	add	br	0	6.63
baking powder	remove	center	0	6.32
cheese	remove	br	0	6.47
cereal	add	t	0	9.35
cereal	remove	t	0	6.75
cheese	add	b	0	6.75
cheese	remove	b	0	6.62
milk	add	t	0	7.04
milk	remove	t	0	6.56
cereal	add	t	0	7.2
cereal	remove	t	0	6.99
cereal	add	right	0	6.91
spaghetti	add	tl	0	6.75
cereal	remove	right	0	6.6
spaghetti	remove	tl	0	6.4
cracker	add	bl	0	6.44
cracker	remove	bl	0	6.47
cracker	add	b	0	6.67
cracker	remove	b	0	6.82
cracker	add	bl	0	6.48
cracker	remove	bl	0	6.416
cereal	add	b	0	7.06
cereal	remove	b	0	7.27327
applesauce	add	br	0	7.39
applesauce	remove	br	0	6.9
tomato Can	add	right	0	6.65
tomato Can	remove	right	0	6.57
cracker	add	br	0	7.1
cracker	remove	br	0	6.55
applesauce	add	b	0	6.48
tomato Can	add	center	0	6.44
applesauce	remove	b	0	6.35
applesauce	add	br	0	6.449
tomato Can	remove	center	0	6.52
milk	add	Left	0	6.87
cheese	add	tr	1	8.528
milk	remove	Left	0	6.75
cereal	add	tl	0	6.72
cereal	remove	tl	0	6.79
applesauce	remove	right	0	7.39
applesauce	add	Left	0	6.396
cheese	remove	right	1	6.35
cereal	add	right	0	6.89
cereal	remove	right	0	6.55
cracker	add	tr	0	6.69
cracker	remove	tr	0	6.29
applesauce	remove	Left	0	6.2
cracker	add	center	0	6.75
cracker	remove	center	0	6.34
cracker	add	tr	0	7.87
cracker	remove	tr	0	7.84
cereal	add	right	0	6.96
cereal	remove	right	0	6.94
cracker	add	right	0	6.75
cracker	remove	right	0	6.45
spaghetti	add	t	0	7.27
spaghetti	remove	t	0	6.5
yogurt	add	b	0	6.74
yogurt	remove	b	0	6.7
cereal	add	Left	0	8.08
cracker	add	tr	0	7.16
cracker	remove	tr	0	6.66
cheese	add	br	0	6.61
cereal	remove	Left	0	7.77
beans	add	b	0	6.82
baking powder	add	Left	0	6.9
beans	remove	b	0	6.69
cheese	remove	br	0	6.69
milk	add	center	0	7.24
milk	remove	center	0	7.75
cracker	add	tr	0	6.73
baking powder	remove	Left	0	6.54
spaghetti	add	bl	0	6.38
tomato Can	add	b	0	6.63
cracker	remove	tl	0	6.44
yogurt	add	t	0	6.48
spaghetti	remove	Left	0	6.62
spaghetti	add	right	0	6.62
tomato Can	remove	b	0	6.72
beans	add	center	0	6.98
yogurt	remove	t	0	6.94
beans	remove	center	0	6.45
milk	add	t	0	7.28
yogurt	add	bl	1	6.91
milk	remove	t	0	6.91
yogurt	remove	bl	0	6.33
spaghetti	remove	right	0	6.66

TABLE IX. Failed Trial (~50 images)

Item	Action	Location	Failed	Time
cheese	add	l	0	7.4
cheese	remove	l	0	6.4
cheese	add	tr	1	6.6
cereal	add	tl	0	6.69
cereal	remove	tl	0	7.4
cheese	remove	tr	1	6.56
crackers	add	c	0	6.5
crackers	remove	c	0	6.7
cereal	add	tl	0	7
cereal	remove	tl	0	6.6
yogurt	add	c	0	6.5
yogurt	remove	c	0	6.2
crackers	add	r	0	6.4
crackers	remove	r	0	6.6
baking powder	add	r	1	6.75
spagetti	add	l	0	6.97
spagetti	remove	l	0	6.85
spagetti	add	t	1	6.62
baking powder	remove	r	1	6.23
applesauce	add	tr	1	7.27
applesauce	remove	tr	1	6.1
applesauce	add	r	1	6.2
applesauce	remove	r	0	6.4
spagetti	remove	t	0	6.47
cereal	add	r	0	6.86
cereal	remove	r	0	7.478
beans	add	tr	1	6.3
beans	remove	tr	0	6.55
baking powder	add	tr	1	6.7
spagetti	add	l	0	6.7
baking powder	remove	tr	0	6.67
baking powder	add	t	1	6.4
spagetti	remove	l	0	6.53
tomatoes	add	tl	1	6.6
tomatoes	add	tl	1	6.6
yogurt	add	tl	1	6.7
baking powder	remove	t	1	6.9
beans	add	t	1	7.3
applesauce	add	br	1	6.88
yogurt	remove	tl	1	6.6
beans	remove	t	1	6.7
applesauce	remove	br	1	6.8
tomatoes	add	l	1	6.33
milk	add	t	0	8.34
tomatoes	remove	l	0	6.78
tomatoes	add	l	1	6.87
milk	remove	t	0	6.5
yogurt	add	b	1	6.5
yogurt	remove	b	1	6.7
tomatoes	remove	l	1	6.3
yogurt	add	bl	1	6.2
tomatoes	add	b	1	6.5
beans	add	b	0	6.49
applesauce	add	br	0	6.45
beans	remove	c	0	6.45
applesauce	add	br	0	6.45
cheese	add	bl	0	6.57
spagetti	add	t	0	6.8