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, Computer Vision, Localization, OpenCV, SIFT**

I. Introduction

IN the hustle and bustle of the modern world, oftentimes it can be difficult 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, both 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, this device is an inventory tracking system. It will maintain an automatically generated inventory of items in the user's kitchen using a Raspberry Pi 3 with an embedded camera and cloud-side computer vision. This hardware will sit on the ceiling of a storage area like a fridge or a cabinet and capture 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 but also *extremely* overly thorough for the average consumer in a home kitchen. App-based tracking systems on mobile devices, on the other hand, 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 automation. As such, we hope to create a system that reduces the user burden by using computer vision to identify items and automatically catalog them.

II. Use-Case Requirements

Ease of use: Given our market niche, from a usability perspective we would like our project to 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. Further breakdown of expected response time values can be found in section IV.

Item Handling: Our project should be able to handle both supported and unsupported grocery items. While we will try to support a number of common grocery items, different users with diverse culinary preferences will 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 will be notified of any 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 Table I). It is especially

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%.

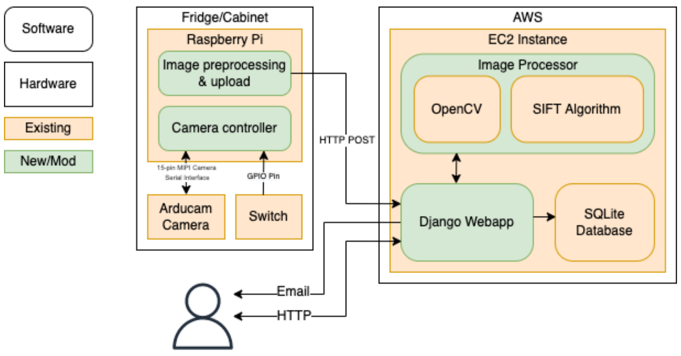


Fig. 2. System block diagram

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 will be able to add their own recipes to their personal cookbook, and the system will automatically check whether the user has all of the necessary ingredients for a given recipe. If there are any ingredients missing, the system will automatically generate a grocery list, which the user will be able to access. Relatedly, the system will support sending the user a grocery list with the said ingredients.

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 will capture an image of the storage area when the user closes the door, and send it to a cloud server running our computer vision algorithm. The algorithm will try to identify the item added/removed by the user. If the item can be successfully identified, the server will update the user's inventory in the database. Otherwise, it will notify the user and ask them to label the item manually. The user will be able to view their inventory or label unidentified items through a web application.

A. Hardware

The hardware unit will consist of an RPi attached to a wide-angle camera module. We expect to be using a low-distortion wide angle M15 lens, though this may vary depending on our performance results while testing. The camera will be placed at ~45 degrees relative to the items in the storage location, to ensure optimal field of view of an item. The exact angle of the camera relative to the items will be determined during testing.

B. Hardware to Cloud

The hardware component will communicate with the cloud deployment with a POST request containing the device's ID, a serialized image, the serial number of the RPi, and a secret validation string. Each of the hardware units will be given a secret validation string, so that a malicious actor can't impersonate another device and post fake JSON with just the serial number. For security reasons the POST request will be encrypted, and the server will verify the correctness of the string via its internal mapping before processing the rest of the request.

C. Computer Vision Algorithm

From here, our cloud deployment of our CV algorithm will identify the item from the image sent. For the computer vision component, we will be using SIFT, as it was found to be the most effective of the algorithms which we tested. For a more comprehensive comparison of the potential advantages and disadvantages of the available algorithms, see section V (Design Trade-Offs).

D. Cloud Deployment

Both our CV algorithm and our web application (web-app) will 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 running two separate instances.

E. Web Application (Web-app)

As mentioned above, the web-app will run on the same EC2 instance. Users will be able to log in, view their registered devices and inventories of those devices, add and modify recipes that the user has stored, and generate shopping lists based on the ingredients the user is missing. All of the supported items will be stored in a database with quantity currently in the fridge and descriptors of the iconic images.

The system will also notify the user if/when an unsupported item is placed in the cabinet, and prompt them to tag the item in as few words as possible.

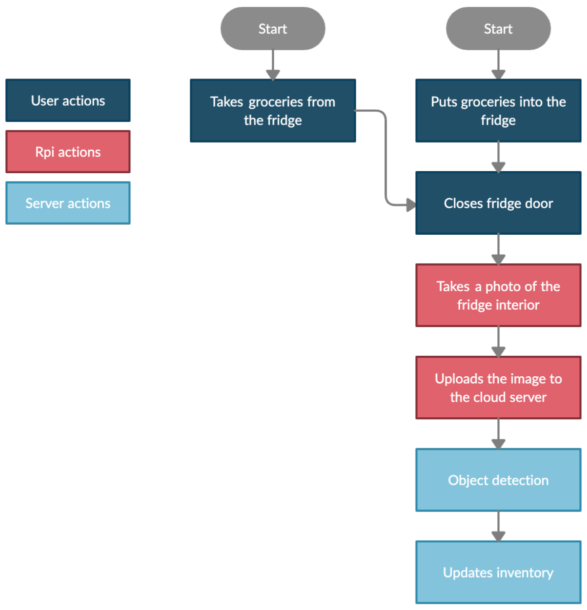

User actions
Rpi actions
Server actions
Start
Takes groceries from the fridge
Start
Puts groceries into the fridge
Closes fridge door
Takes a photo of the fridge interior
Uploads the image to the cloud server
Object detection
Updates inventory

Fig. 3. User actions flow chart.

IV. Design Requirements

For our design requirements, most of the requirements we discussed in the use-case requirements section 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 see no other way if we want 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, it is needed to satisfy a different use case requirement, the ability to handle unsupported items. This will enable us to perform pixel differentiation to localize any object that we do not support. This will also help us with localization in general, which will likely make 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 more 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 will be doing 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 have decided to base our update time requirement on the worst case scenarios of the CV component; We allow for no more than 7 seconds from item placement to website update.

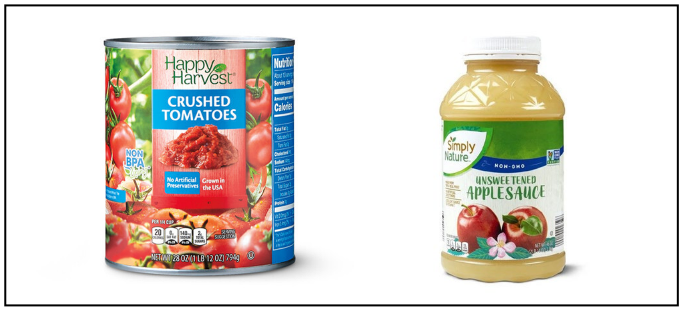


Fig. 4. The two items with longest feature matching. Both items took about 4.5 seconds to extract features.

V. Design Trade Studies

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

A. *SQLite vs. Alternatives*

SQLite generally is less effective when handling large numbers of transactions concurrently. However, our project will not generate a large number of transactions—we estimate ~10 per user per day. Additionally, we are using Django for the webapp, 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 which we would use for object detection and classification, we looked at SIFT, ORB, BRIEF, YOLO, and other neural network based classifiers. 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 would work for our purposes, so the amount of effort that would be required to generate the required training data would likely be substantial. Finally, if we used a neural network based algorithm, we would be 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. 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 angle may vary, and the final images will have some lens distortion, and the focus may not be as good. However, we feel that this test was effective for the purpose of comparing the given algorithms.

As shown in Table 3 (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 ~10 times a day. Although we can get lower latency, the computational resources of the Jetson Nano will 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: 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.
- 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. This will not make financial sense if the product is to be commercialized.

Therefore, we felt it was difficult to justify the cost of having dedicated local hardware, and we decided to do the computer vision processing remotely.

D. *RPi vs. Alternatives*

There were several possibilities for the hardware we could use. 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* 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 will 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. Alternatives*

Members of our team have experience using Django, and there's no specific features offered by another framework which are needed for our use case. Therefore, we've decided to just use Django as the framework of choice.

VI. System Implementation

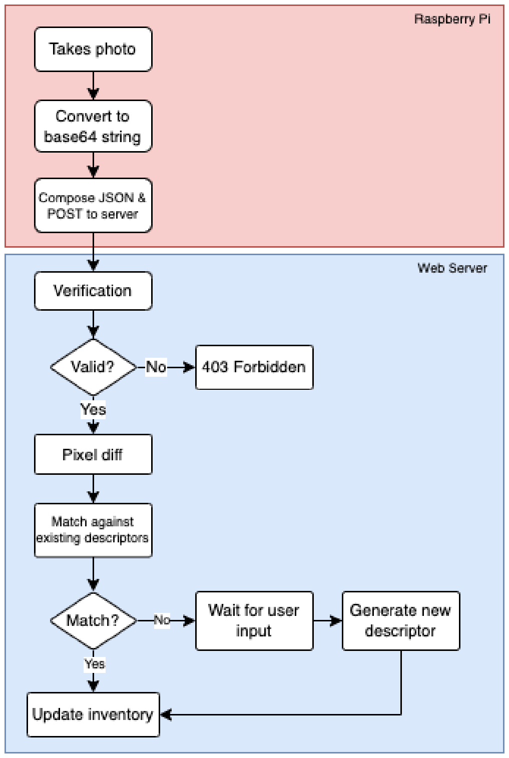


Fig. 5. Computer Vision flowchart.

A. *Hardware*

The hardware component will consist of a Raspberry Pi (RPi) model 3B, the OV5647 Arducam camera module, and a button.

The RPi 3B is the cheapest model available that has built-in wifi connectivity, and an MIPI Camera Serial Interface Type 2.

Hypothetically, we would expect this sensor unit to be bundled with a smart storage appliance, but for the scope of this project this is not feasible. Therefore, the door to the storage area closing will be simulated by the press of the button connected to GPIO pin 5, and the RPi will be supplied power directly from a wall socket.

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. This lens can be detached and replaced with a different lens. We intend to test several lenses before deciding which one works best. (Due to shipping delays, we did not have an opportunity to test before this design report).

Each RPi will know its own serial ID, and its secret string, which will be used when communicating with the web-app. The sensor will sit idle until it reads a button press on GPIO pin 5. When this happens it will take a photo, and send it to the web-app by an HTTP POST, before returning to idle. See Section C on communication for details between the web-app and the hardware component.

B. Web-app

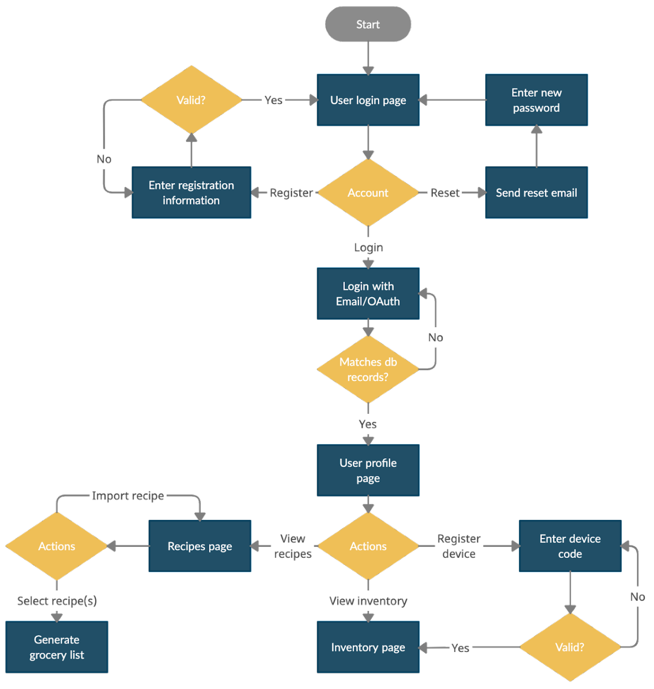


Fig. 6. Website flow chart

The web-application will be hosted on an AWS EC2 instance. Its port 80 will be enabled to allow incoming HTTP traffic.

We will use the Django framework to implement the web-app, handling any incoming requests from the hardware components as well as displaying the inventory information to the user. We will use the default database SQLite 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 will use model classes for the creation of different objects like users and devices. A class diagram can be found in the Appendix (Fig. 9).

When the user first visits the web-app, a registration page will be prompted. User authentication will be implemented using the python-social-auth library, which supports logging in from a number of different services through the OAuth protocol, including Google, Facebook, Twitter, etc.

After the user is authenticated, the web-app will ask for a device serial number to finish the registration process. We will maintain a list of all the devices we "manufacture" to prevent fraudulent device registration. When the user enters the serial number of their new device during registration, we will activate this device and assign the current user as its owner.

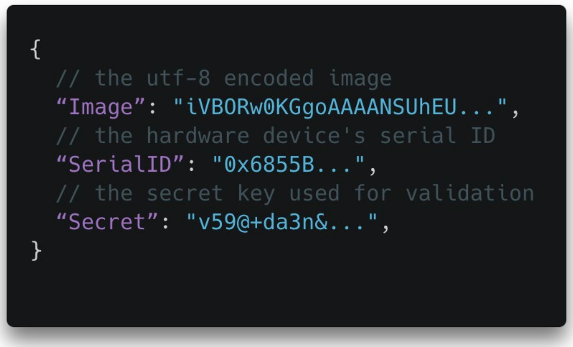
When handling incoming POST requests, the web-app will invoke the CV component for object classification. If the CV component is able to classify the object, the web-app will update the inventory in the database accordingly. Otherwise, the web-app will send the user an email through the Gmail SMTP server. The web-app will display a thumbnail image of the object obtained from pixel difference calculations in the CV component and ask the user to manually identify the item.

Each identified item will be assigned a location field to indicate the device that currently holds the item. This is useful if the user owns multiple devices, and if this is the case, the web-app will be able to display the inventories separately. This will be done by filtering item entries by devices owned by the user. The web-app will also be able to display a combined inventory, which will be useful if the user requests a shopping list.

C. Communications

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

When the device takes a photo, it will convert it to a string of utf characters using python's base64 package. It will then send the request with the information in Fig. 7:



```
{
  // the utf-8 encoded image
  "Image": "iVBORw0KGgoAAAANSUhEU...",
  // the hardware device's serial ID
  "SerialID": "0x6855B...",
  // the secret key used for validation
  "Secret": "v59@+da3n&...",
}
```

Fig. 7. An example JSON message.

The web-app will only accept requests coming from activated devices. It will also store the secret key for each device in the database to validate requests. Using the secret key ensures that a malicious actor will not be able to modify someone else's inventory by simply modifying the serial number in the request.

The hardware device will also send POST requests with an empty "image" JSON field periodically (every 10 minutes) as heartbeat messages to the server to indicate its online status. If the server hasn't received any messages from an activated device for 30 minutes (missing three heartbeats), its owner will be notified through email.

When the user is visiting the website, the client-side JavaScript code will use AJAX to pull information from the server 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, 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 will extract the features and key points from the iconic images for each of the globally registered image classes. The grocery items that will be 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 will then be held in some global state variable, and will be used until the web server shuts down.

When invoking the computer vision code, the web application will pass the following information: image of previous state, image of new state, and a map of locally registered item categories to their descriptors. The CV component will first perform a pixel differentiation between the image of the previous state and the image of the new state to localize any regions of change.

First, we will run SIFT to extract the keypoints and descriptors from the region of interest in the new state image. We then attempt to match the descriptors to each of the known grocery items' descriptors. If we successfully identify any grocery items during this step, it means that the user has added that grocery item to the storage location. If we fail to find any items of interest, we repeat the above process on the region of interest in the old state image. If we successfully identify any grocery items during this step, 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, and the user is prompted to name the new grocery item category. The newly created category and its descriptors are then added to the user's map registered grocery items.

VII. Test, Verification and Validation

To simulate updating a kitchen inventory, we'll subdivide two storage locations and number the available slots. We will then pick a random location and perform one of two actions. If it's occupied, remove the item. If it's unoccupied, fill it with an item randomly chosen from our accepted grocery list. This process will be used for the testing procedures below.

A. Response Time

In order to test update speed (i.e. time from button press to website update), we will create a mock page that is identical to the regular inventory page that also displays the linux time since when the last update occurred. We will also have the RPi print the linux time when the button is clicked. By taking the difference, we will have the total response time of the system. This testing can be done in conjunction with the classification accuracy test.

B. Classification Accuracy

To check classification accuracy, we will repeatedly update the kitchen's inventory, keeping track of the success, failures, and misidentifications per item. We will repeat until we've seen every item 10 times or we have 100 trials, whichever takes longer. In conjunction with the response time requirements, this testing setup will allow us to check both requirements simultaneously.

C. Heartbeat Testing

To test the heartbeat mechanism, we will unplug a router for 45 minutes to ensure that one notification is sent after 30 minutes of inactivity, and no more than one is sent erroneously.

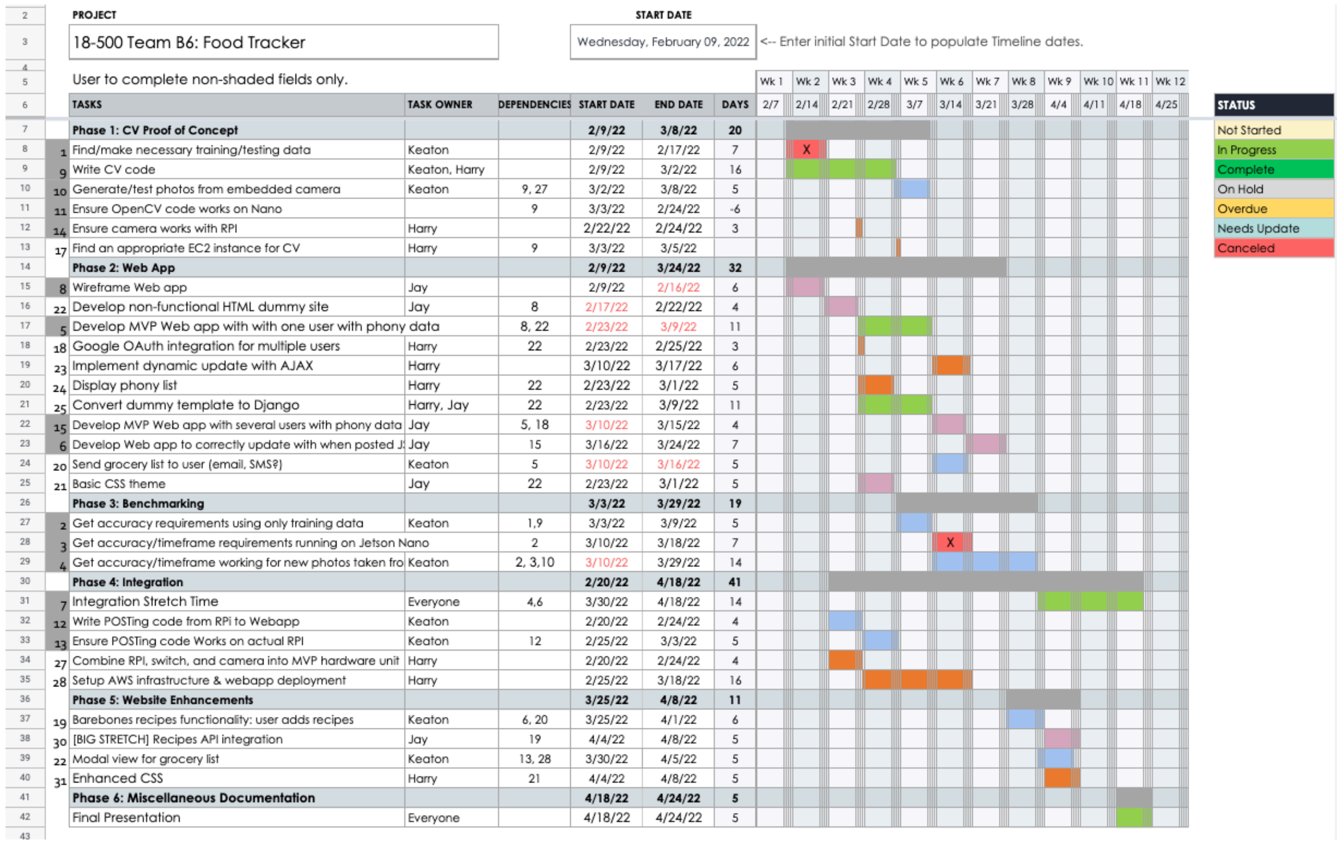
VIII. Project Management

A. Schedule

We've divided our Gantt chart into subsections (Fig. 8): 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.

B. Team Member Responsibilities

Generally speaking, Keaton is primarily responsible for writing and testing the CV used for object classification. Harry is secondarily responsible for this, he will assist with this when he is able. Harry is primarily responsible for the web-app backend and communication between the hardware sensor and the web application. Jay will likely help with this, especially as it pertains to his primary work with the front end. Jay is primarily responsible for the web application front end and assisting the backend.


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.

	TASKS	TASK OWNER	DEPENDENCIES	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	9, 27	3/2/22	3/8/22	5
11	Ensure OpenCV code works on Nano		9	3/3/22	2/24/22	-6
14	Ensure camera works with RPI	Harry		2/22/22	2/24/22	3
17	Find an appropriate EC2 instance for CV	Harry	9	3/3/22	3/5/22	
	Phase 2: Web App			**2/9/22**	**3/24/22**	**32**
8	Wireframe Web app	Jay		2/9/22	2/16/22	6
22	Develop non-functional HTML dummy site	Jay	8	2/17/22	2/22/22	4
5	Develop MVP Web app with with one user with phony data		8, 22	2/23/22	3/9/22	11
18	Google OAuth integration for multiple users	Harry	22	2/23/22	2/25/22	3
23	Implement dynamic update with AJAX	Harry		3/10/22	3/17/22	6
24	Display phony list	Harry	22	2/23/22	3/1/22	5
25	Convert dummy template to Django	Harry, Jay	22	2/23/22	3/9/22	11
15	Develop MVP Web app with several users with phony data	Jay	5, 18	3/10/22	3/15/22	4
6	Develop Web app to correctly update with when posted J:	Jay	15	3/16/22	3/24/22	7
20	Send grocery list to user (email, SMS?)	Keaton	5	3/10/22	3/16/22	5
21	Basic CSS theme	Jay	22	2/23/22	3/1/22	5
	Phase 3: Benchmarking			**3/3/22**	**3/29/22**	**19**
2	Get accuracy requirements using only training data	Keaton	1,9	3/3/22	3/9/22	5
3	Get accuracy/timeframe requirements running on Jetson Nano		2	3/10/22	3/18/22	7
4	Get accuracy/timeframe working for new photos taken fro	Keaton	2, 3,10	3/10/22	3/29/22	14
	Phase 4: Integration			**2/20/22**	**4/18/22**	**41**
7	Integration Stretch Time	Everyone	4,6	3/30/22	4/18/22	14
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	12	2/25/22	3/3/22	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
	Phase 5: Website Enhancements			**3/25/22**	**4/8/22**	**11**
19	Barebones recipes functionality: user adds recipes	Keaton	6, 20	3/25/22	4/1/22	6
30	[BIG STRETCH] Recipes API integration	Jay	19	4/4/22	4/8/22	5
22	Modal view for grocery list	Keaton	13, 28	3/30/22	4/5/22	5
31	Enhanced CSS	Harry	21	4/4/22	4/8/22	5
	Phase 6: Miscellaneous Documentation			**4/18/22**	**4/24/22**	**5**
	Final Presentation	Everyone		4/18/22	4/24/22	5

Wk 1 Wk 2 Wk 3 Wk 4 Wk 5 Wk 6 Wk 7 Wk 8 Wk 9 Wk 10 Wk 11 Wk 12
2/7 2/14 2/21 2/28 3/7 3/14 3/21 3/28 4/4 4/11 4/18 4/25
X
X

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

Fig. 8. Gantt Chart with milestones and team responsibilities

C. *Bill of Materials and Budget*

(Table II) Overall, we have used a very small amount of our budget, and I do not expect us to need to use any more in the foreseeable future. The only likely additional expenses are different lenses.

D. *Risk Mitigation Plans*

Most aspects of our project can be done independently, with little to no risk of catastrophic failure. We view 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, there are several values which can be tweaked that can affect the overall accuracy of the system. Namely, the confidence threshold on the SIFT algorithm, and the threshold on the pixel difference algorithm.

Given our results from Part V, it seems unlikely that we can switch algorithms to either ORB or BRIEF (both of which are faster). However, there are several algorithms which were not tested which may provide comparable results at higher speed, namely SURF. Additionally, it may be possible to speed up the overall response time by using a higher capacity EC2 instance.

IX. 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 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 the 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. 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.

X. Summary

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 to provide a successful tracker that will fulfill the grocery-shopping needs of every home cook.

Glossary of Acronyms

AWS	Amazon Web Services
BRIEF	Binary Robust Independent Elementary Features
CV	Computer Vision
EC2	Amazon Elastic Cloud Compute
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
OAuth	Open Authorization
ORB	Oriented FAST and Rotated BRIEF
RPi	Raspberry Pi
SIFT	Scale Invariant Feature Transform
SMTP	Simple Mail Transfer Protocol
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/.

Table II. Bill of Materials

Description	Model #	Manufacturer	Quantity	Cost @	Total
RPi 3 Model B	4328498196	RPi Foundation	2	$100	$200
1/2.7" 2.8mm Focal Length Lens		ARDUCAM	2	$18	$36
OV5647 with M12 Lens preattached		ARDUCAM	2	$28	$56
AWS Credit		Amazon	1	$50	$50
				Grand Total	**$342.00**

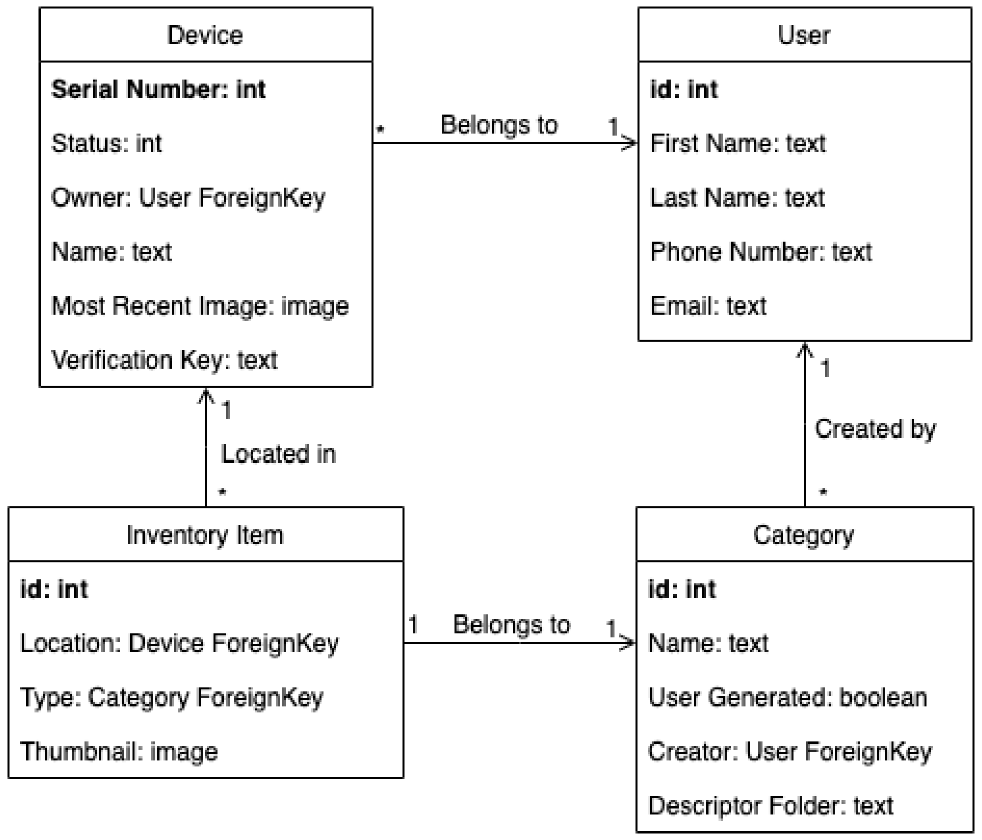


Fig. 9. Class Modal Diagram of the Database

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 III. COMPARISON OF VARIOUS CV ALGORITHMS

The x-axis is the iconic image, and the y-axis is the actual item.

1.000 represents a 100% match between the actual item and the iconic image.