FreshEyes

Authors: Alex Strasser, Oliver Li, Samuel Leong
Affiliation: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—FreshEyes is a smart fridge attachment and interface system that tracks fresh produce using computer vision and AI. Our user-friendly and integrated system can be placed on the door of any fridge, and users simply scan fruits and vegetables by placing it in front of our smart camera system. Our intelligent interface additionally prompts the user about potentially-expiring produce, and sends them a weekly summary what they have consumed. It will even suggest some recipes that utilize expiring foods.**

***Index Terms*—Computer Vision, Databases, Inventory Tracking, Mobile App, Smart Fridge**

1 INTRODUCTION

According to the Environmental Protection Agency [1], the United States alone wastes approximately 40 million tons of food per year (approximately 219lbs per person). Based on the weighted median of fruit and vegetables per pound calculated by the US Department of Agriculture [2], this amounts to approximately $219 \times \$1.50 = \textit{\$328.50}$ of money wasted per person per year! In households, most food is thrown away because they are forgotten and left to rot or expire in fridges. We aim to address this issue with our solution, FreshEyes, targeted at households who shop for fresh produce (such as eggs, milk, fruits, vegetables etc.). Our proposed user-friendly and integrated system can be placed on the door of any fridge, and combines computer vision with an intuitive user-interface system to non-intrusively track fresh produce going in and out of the fridge. The intelligent system prompts the user about potentially-expiring produce, and will even suggest some recipes that utilize said foods.

2 USE-CASE REQUIREMENTS

Given the problem of households discarding expired produce, we aim to reduce the amount of food waste per person by 1 fruit and vegetables a week. This would result in approximately $1.50 saved per week (the approximate cost of a produce item), and approximate 25 pounds of food waste per year.

To this end, assuming a person buys 7 items in a week, we want to only misdetect, on average, one of those items. This means we need an 85% scanning accuracy. We also want to be able to differentiate at least 10 items, since this will, at bare minimum, cover the common items you would buy over a couple weeks (more than 7). We want each scan to take 2 seconds on average to show up in the user interface (UI), and each UI interaction to take 2 seconds on average. This results in a total of 28 seconds per week spent scanning.

Additionally, since this product is meant to be a modular system that can be added onto your existing fridge, we want it to be easy to install and uninstall. We do not want to require any permanent modifications to the fridge for installation, and both installation and uninstallation should take under 2 hours. In order to be time effective for the user, we will also require less than an hour per year (on average) spent maintaining the system (including changing batteries, untangling cables, etc) so that the interaction with the system is hassle and frustration free.

Figure 1 below summarizes the cost and time justification of our use-case requirements: Using the above metrics, we calculated an estimated cost of 30 seconds for 7 items (2 seconds each for scanning and UI interactions) and saving an average of $1.50 per week. However, because the median salary of workers in Pennsylvania is approximately $0.50/min [3], we have an estimated user savings of a $1.25 per week. While this might not seem like much, this amounts to $65 a year, and also helps save the environment by reducing food wastage!

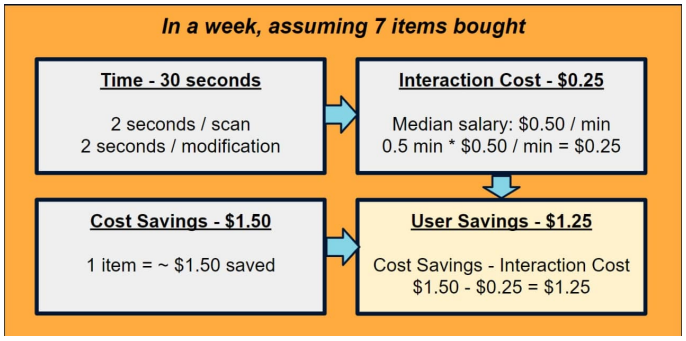


Figure 1: Visual Summary of Use-Case Justification

3 ARCHITECTURE AND/OR PRINCIPLE OF OPERATION

Our architecture consists of 4 primary components/-subsystems, whose relationships are briefly summarized in Figure 2 below, and in greater detail in Figure 6.

1. Computer Vision System (CV)
2. Database System (Back-End)
3. User Interface (Front-End)
4. Integrated Fridge-Attachment System

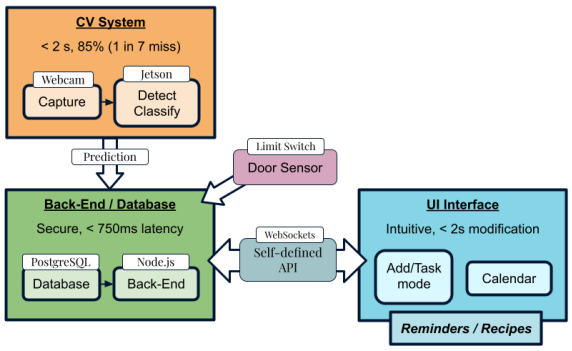


Figure 2: Summary of Architecture. Detailed diagram in Figure 6 below. Big rectangles represent key subsystems. Smaller, colored rounded rectangles represent key actions or components, with the uncolored rectangles above them being the implementations or hardware used.

3.1 Computer Vision System (CV)

The computer vision (CV) system's job is to capture, detect and classify the items presented to it by the user. It will first capture the item (eg. a fruit or vegetable) using a webcam; the image is then passed on to our Jetson Nano, which will then detect the item and classify it using the image processing algorithms outlined in Section 6.2 below. The image processing algorithm will then spit out a set of top 4 predictions, which it then sends over to the Database System (Back-End) via our self-defined Application Peripheral Interface (API).

3.2 Database System (Back-End)

The database system is the backbone of our entire system, processing the requests from the Computer Vision System (CV) and the User Interface (Front-End), and sending relevant information over to the User Interface (Front-End) where it is presented in a user-friendly way. Conversely, user input such as manual quantity adjustments are sent from the User Interface (Front-End) to the back-end, where it is processed and stored. The back-end is responsible not just for data retrieval and storage between the database, but is also responsible for processing the data before sending it over the User Interface (Front-End). In other words, the User Interface (Front-End) should not need to do any further calculations before the data can be displayed. All interactions between the back-end and the other aspects of the system will be implemented as an API endpoint, allowing for maximum modularity and even allowing for alternative implementations altogether.

3.3 User Interface (Front-End)

The user-interface (UI) serves 2 primary purposes: allow the user to interact with the scanning system, and providing value to the user by sending them reminders of expiring items and potential recipes making use of said produce. In order to do this, it communicates with the database back-end primarily via Websockets, sending inputs by the user and retrieving processed information from the Database System (Back-End). Specifically, the scanning user-interface will allow users to add/remove items from the fridge, and confirm/modify results of a CV scan; the calendar and reminder app interface allows the user to view their expiring produce and any intelligently-suggested recipes utilizing them.

3.4 Integrated Fridge-Attachment System

We then incorporate all of the crucial software and hardware components detailed above onto a large integrated board, which holds our webcam and tablet, as well as a little (foldable) platform for the user to place their items for scanning. There is also a door sensor (in the form of a limit switch) that tells the back-end when the door is being opened or closed, facilitating more efficient tracking of produce. Our integrated board allows for the user to easily install (or uninstall) our system as a module on any fridge door. Notably, we should not forget that the server back-end and the user's mobile device (if they so choose to use our application on it) is part of the larger integrated system as well; both hardware and software are integrated together seamlessly for an optimal user-experience. A diagram of how the *physical* integrated system will look like is depicted in Figure 3 below.

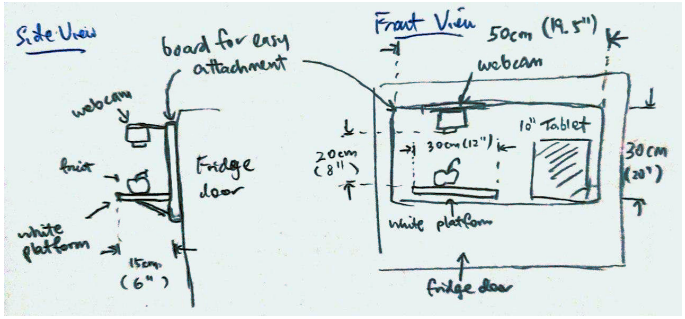


Figure 3: Front and Side View of Attachment System

More details on the implementation of said architecture can be found in Section 6 below.

4 DESIGN REQUIREMENTS

Most of the justification for the design requirements is found in Section 2.

As noted in Section 2, We need our CV segmentation and classification system to detect and classify objects with more than 85% accuracy, if we are to achieve the desired 6 out of 7 item matches. This classification time, plus processing time on the database should take under 2 seconds.

In order to support the CV system effectively, which requires a 100x100 RGB image as an input to the neural

network, we need an RGB camera that has a minimum resolution of 500x500, with a picture fetching time of under 250ms to adequately satisfy timing constraints.

The back-end/database system must be very robust so that it does not require any human interaction. It must be able to handle up to 3 clients simultaneously with no performance degradation. It should not lose track of any data.

The user interface needs to be very responsive. It should have a response time of under 200ms for all simple operations (changing the mode or quantity) and under 750ms for complex operations (submitting a modified scan to the backend). This 750ms response time is also a requirement for the backend.

5 DESIGN TRADE STUDIES

5.1 Barcode Scanner

One of our initial ideas was to use a barcode scanner to scan items, as this makes detection a lot easier. This is, after all, how object detection is done in grocery stores. However, this requires a large barcode database and generally works very poorly for the produce items we want to generally track. Bunches of cilantro, for example, very frequently do not have a barcode and rely on a product code at the grocery store.

Additionally, this can be very difficult for the end user. It takes time to find the barcode and orient the item such that the barcode can be read. This would not be able to meet the 2 second average scan time, even though it would pass the accuracy requirement.

5.2 BLE / NFC tags

This design would also make the detection and tracking of items much easier. One of the main issues with the tracking system was detecting when items would leave and return to the fridge, instead of a new item just being bought. However, this requires associating tags with produce items and offloads all the classification processing to the user. This ends up being a lot more work for the user and would likely meet the 2 second average scan time, but would require enough set up and pre-scan work that it would be ineffective.

Additionally, the tags themselves are not very reusable and the cost ends up being prohibitively expensive. At around $0.50 per tag, and 7 items per week, we would spend $3.50 on tags and only save $1.50 on groceries. Tags could theoretically be reused, but the stickiness won't last and the tags would not be able to track items being returned vs purchased (determining whether the tag is on a new item or old item).

5.3 Cameras Inside the Fridge

Our current solution involves an external scanning-based approach where the user manually scan the items with a camera mounted *outside* the fridge. An alternative solution that we considered was to have cameras *inside* the fridge that would detect and classify the items placed inside the fridge. This would have been an ideal solution for user-intuitiveness because it does not disrupt the normal workflow of one's normal loading/unloading of groceries. However, this solution suffers from 2 major issues: firstly, occlusions are almost unavoidable since people tend to stack items in the fridge, constantly rearrange items within the fridge, and there is no camera angle for which all items can be seen properly; secondly, cameras and other sensors do not usually perform well in constant cold, and having a modular system would either need external wiring that would affect insulation, or battery-operated systems that would need constant recharging or replacing. To resolve Problem 1, one could use multiple cameras, but this would increase complexity (since we would need to detect which objects are seen in both cameras) and still does not resolve issues regarding occlusions from stacking or objects hidden inside plastic bags. Problem 2 can potentially be resolved or mitigated if modularity was no longer a requirement: the cameras could be custom-made to withstand cold temperatures, and wiring integrated as part of the fridge design itself, but this might be a prohibitively expensive thing to do, and would require users to buy an entirely new smart fridge. This is wasteful design and would be counterproductive with our goal of saving cost and reducing environmental impact.

5.4 Fully Local System

We also considered implementing a fully local, non-Internet facing version of our proposed system. Such a system will bring the computer vision system, front-end, the back-end, and the database into one local monolith. While such a system will be significantly less complex and also avoids the round-trip time latency between the CV system, back-end, and the front-end, it also means fewer possible features and less room for future expansion.

This project was built with the ability to access the front-end from any Internet connected device in mind. For example, a user will want to look up their fridge inventory while they are doing their grocery shopping. A fully local system will mean that such accesses will not be possible.

Furthermore, we also designed the project with a view towards future expansion and stretch goals. One such feature we envisioned is a shopping list that can be shared among family members, where each family member can view what is available in the refrigerator from any Internet connected device, and add their own items to a shopping list. Such features are only viable in an Internet connected system, with a front-end built upon HTML and other standard web technologies, and a separate back-end.

5.5 Heuristic-based CV Algorithms

Specific for the CV system, we considered using a heuristic-based detector and classifier instead of a CNN-

based segmentation and classification system. In particular, we considered using a classical ORB + Bag of Words method for detection and classification respectively. This would allow us to potentially get rid of our white platform setup, and make for a more intuitive experience with a basic front-facing camera that the user can hold a fruit or vegetable up to. However, it is likely that such classical might suffer in terms accuracy, which would then negate any intuitiveness from the scanning process by adding on the inconvenience of having to continually correct the algorithm's predictions. Moreover, given the advances in learning-based CV methods, we decided that the increased robustness and accuracy that such systems would provide would be a better trade-off. This was also affirmed with our initial tests on the classifier which provided us with 98% testing accuracy. As mentioned above, the only issue is that our classifier is being trained on data with a white background, and therefore an accurate segmentation algorithm is required. However, our white platform setup (see Figure 3) will effectively mitigate this problem, and also allow us to use a simple heuristic-based algorithms (pixel comparisons, thresholding, floodfill) for detection and segmentation.

6 SYSTEM IMPLEMENTATION

6.1 Integrated Fridge-Attachment System

With reference to Figure 3 above, the integrated attachment system will have base made out of a piece of 15cm $\times$ 30cm (approx $6'' \times 20''$) of solid plywood, onto which our tablet and webcam setup will be stuck. The plywood backing is then intended to be installed on the front of a fridge door with simple 3M mounting tape.

Notably, our webcam setup involves a web camera pointing downwards towards a white platform. The white platform is necessary for the proper working of the Computer Vision System, with more details in that section. The placement of the web-camera and platform is crucial for the unobtrusive use of the scanner; if we were to have the camera face forward with a vertical platform (as originally designed), our users would have to awkwardly place the fruit from above, which would be quite very obtrusive.

6.2 Computer Vision System

The computer vision system will first obtain an image using the webcam. Notably, as shown in Figure 3, the webcam is setup such that it is pointing down towards a white platform, resulting in the received image being a simple off-white image when idle. Therefore, by simply checking for significant changes in pixels, we can quickly tell whether a fruit is put on the platform for scanning. A simple floodfill-based segmentation algorithm will then be used to segment the item from the white background. The preprocessed image will then resized appropriately and sent through a ResNet18-based [4] convolutional neural network (CNN), which will output a set of probabilities. The top 4 probabilities are then sent to the back-end via our self-defined API, and will subsequently be displayed to the user for confirmation (default) or correction (if any), thus allowing for an intelligent, semi-automatic tracking process.

Notably, we will be training the CNN ourselves using PyTorch [5] on the Fruits360 dataset [6], which has a total of 131 classes. If necessary, we will collect our own data from our setup to additionally train our network with. However, all image processing algorithms, including the neural network, will be written in optimized C++ code, and will be running on a Jetson Nano for maximum efficiency.

6.3 Database System (Back-End)

The core of the back-end will be built upon Node.js, Express, and TypeScript. These are widely used, industry standard technologies with significant amounts of available documentation and have also been well-optimized for speed. The choice of TypeScript over vanilla ECMA Script (also known as JavaScript) helps catch bugs in development, even before testing, and TypeScript is a type-safe language that enforces strict type-checking even during development. The specific API endpoints, database schema, and architecture of the back-end API is shown in the relevant back-end sections of the block diagram at Figure 6.

The database schema is defined as code (see Figure 4) in a JSON-like format which is ingested by an ORM, specifically Prisma, which creates the tables based on the definition. This allows the schema to be checked into and tracked by our version control system (Git), just like any other piece of code. Furthermore, this also allows us to easily switch out the underlying database system easily between SQLite on development instances and PostgreSQL on production systems. Additionally, the database schema definition also includes the types of each field, allowing database reads through Prisma to have type information as well, enforcing type-safety throughout the back-end.

An API request from a client is first routed to an Nginx web server, which is responsible for negotiating an SSL/TLS connection, and serving any static assets. This is because web servers like Nginx are well optimized for such tasks, allowing for fast response times and easy configuration, lowering the load on the Node.js server. Nginx then forwards any API requests to the Node.js server, where the Express router first routes the request to a API secret key validation middleware, which validates the secret key supplied with the request, dropping any requests with an invalid key. After passing the validation, the request is then routed to the middleware specific to the API endpoint, which queries the database, runs any needed computations, and returns the results to the clients.

This back-end will be deployed on an Internet connected server. This server is a virtual machine running on a physical host, with 1 vCPU and 4GB of RAM. As we are serving a maximum of only 3 clients, this is enough hardware for all back-end components, as they can be scaled to their mini-

```
model Item {
  id           Int           @id @default(autoincrement())
  name         String
  shelfLife    Int // Number of days item can be stored
  unit         String
  transactions Transaction[]
}

model Transaction {
  id        Int      @id @default(autoincrement())
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  item      Item     @relation(fields: [itemId], references: [id])
  itemId    Int
  quantity  Int
  type      Int // 0 = addition to fridge from store, 1 = removal, 2 = addition to fridge (without
    change to exp date)
}
```

Figure 4: Sample of database schema defined in code, with relations explicitly defined

mum possible sizes. For example, the V8 engine powering Node.js can be scaled down to have a maximum memory footprint of just 256MB.

6.4 User Interface (Front-End)

The user interface system will be a web-based interface hosted on the Jetson Nano, as this is our main computer for the project. It will also allow for easy communication to the back-end. which is hosted in the same place. As mentioned in the Database System, the interface client will communicate to the backend via HTTP endpoints and web sockets.

The interface will primarily run on a 10" Android tablet as the client. The tablet will be affixed to the fridge as described in Section 6.1. We selected the MAGCH M101 tablet since it is fast, has a nice screen size, and has enough processing power/camera resolution that some computer vision could theoretically be run on it in the future.

Figure 5 below shows an initial prototype of the UI, also hosted on Alex's server here. Notably, we have designed our scanning interface (see Figure 5b) for maximal intuitiveness and minimal navigation. To this end, we show the top 4 most probable predictions as determined by the CV system, so that the user can make any changes with minimal taps. Selection of multiple quantities can be done with a quick tap, since it would be inefficient to scan a bag of, say, 8 oranges one-by-one. We have also designed the calendar view to be as intuitive and aesthetically pleasing as possible, shown in Figure 5a.

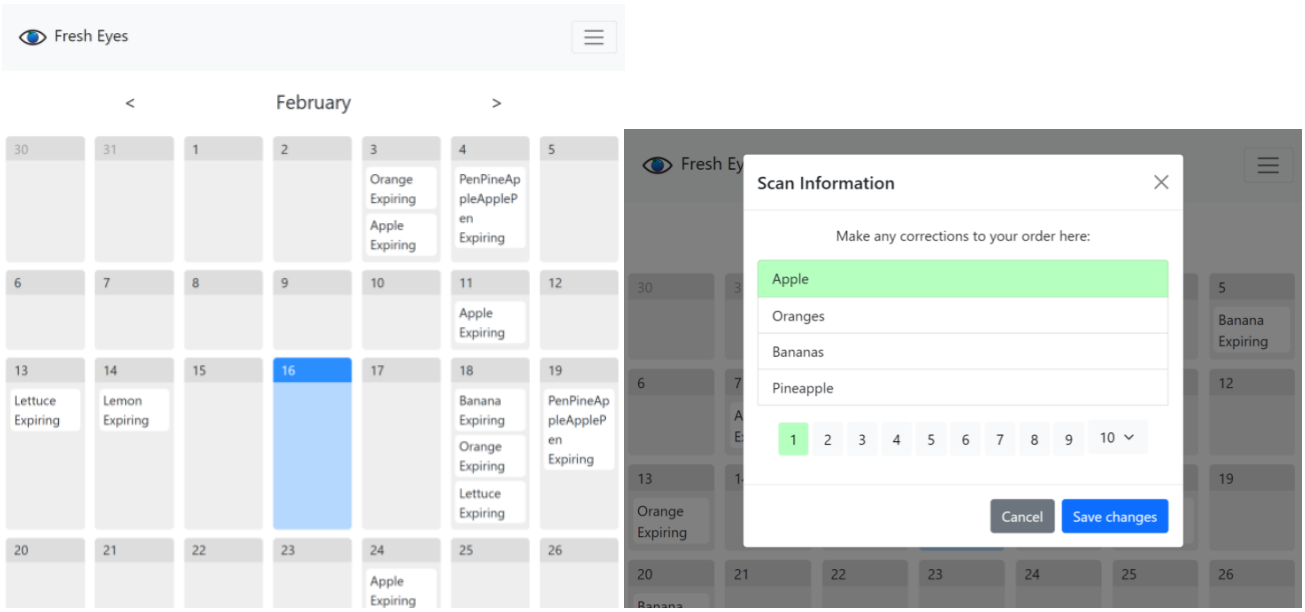
7 TEST & VALIDATION

7.1 Tests for Computer Vision System

The computer vision system needs to be satisfy an accuracy of 85% (1 in 7 items), and less than 2 seconds of latency (i.e. from item being placed for scan, to prediction popping up on the screen). Our first accuracy test will be that of the automated testing accuracy from training the neural network. This is done by splitting the Fruits360 dataset [6] into a training and testing set of images. However, we will aim for a testing accuracy of 95% because we expect our system to perform worse in real life, as the network is not trained on our system-specific data.

Our second accuracy test will be that of a "sanity-check" accuracy test where we will obtain a set of 21 images of common fruits and vegetables (3 each of: apples, oranges, bananas, pears, carrots, lettuce and potatoes). This test is very important to check the generalizability of the neural network. In fact, we were initially ahead of schedule after having trained a network found online. However, we quickly found that the 98% testing accuracy reported by the neural network completely failed at this testing stage. Upon consultation with our professor's PhD student, we realized that the network had severe flaws, and are therefore going to be training our own ResNet18-based [4] network instead.

Finally, we will perform a full-scale speed and accuracy test, where we will purchase a bunch of commonly used fruits and vegetables from the store (apples, oranges, bananas, pears, carrots, lettuce, potatoes, and maybe more), and scan them on our real system with the proper webcam and white platform setup as shown in Figure 3 above. Notably, because of the way our system is designed to be modular, this "real system" will not necessarily involve a real fridge; instead, we will simply test it on our webcam-platform setup on the attachment board. A stop watch will be used to time taken between the fruit being placed on the platform for scanning to when the prediction appears on the screen. The number of correct and false predictions will also be taken down to calculate the overall accuracy later. This experiment will be repeated twice (with the average taken) to ensure reliability.



(a) Calendar View (b) Scan Interface

Figure 5: Initial Prototype of UI

7.2 Tests for API Endpoints

The correctness of API endpoints can be tested during development by sending test API calls using an API testing tool such as Postman. For example, we can set up Postman to first send an API request that gets the number of bananas, followed by a request that adds a banana to the fridge, and finally another request that gets the number of bananas. The number of bananas from the second call should be exactly 1 more than the first call. This checks for the correctness of both the endpoint that gets the quantity of items in the fridge and the endpoint that adds items to the fridge.

Postman also allows for such API calls to be saved and replayed, and even integrated into a testing script that Postman can run automatically from the cloud. This means that we can continuously monitor the uptime of our API, and receive real time alerts when it goes down or returns faulty results, allowing for a high degree of assurance for the correctness of the API as changes are pushed and deployed.

Additionally, this testing process is also how we validate that our metrics such as a maximum latency of 750ms has been achieved. Together with the continuous uptime monitoring, we can view a graph of these metrics from a single dashboard and receive alerts when our metrics are not being met, and even view statistical results such as the tail p99 latency.

8 PROJECT MANAGEMENT

8.1 Schedule

Currently, our project is on schedule and no adjustments are needed to our initial timeline. Our Gantt Chart with the current completion status of each task is shown in Fig. 7. With the submission of this design document, we will have completed our design phase and will be able to fully commit towards the development phase. Some tasks originally scheduled for the development phase, such as hardware ordering, API implementation, and CV training, have already been started even in the design phase.

The planned schedule also includes generous amounts of slack, in the form of time allocated for refinements and revisions. In total, that adds up to about 2 weeks of extra time, significantly mitigating the risks of development overruns from any single part of the project.

8.2 Team Member Responsibilities

All of our members have rough experience in all the subsystems of the project, with some individuals being more specialized than others. Therefore, each of us will be in charge of a specialization, but will also be helping others in their respective sub-tasks. Our team responsibilities are summarized in Table 1 below.

Table 1: Summary of Team Member Responsibilities

Member	Specialization	Helps With...
Alex	Front-End	Back-end, CV
Oliver	Back-End, API	Hardware, Admin
Samuel	Computer Vision	Front-End, Hardware

Table 2: Bill of materials

Description	Model	Manufacturer	Cost
Main Computer	Jetson Nano Developer Kit	Nvidia	$63.50
Tablet	M101	MAGCH	$149.99
Webcam	Webcam HD 1080P	HZQDLN	$21.90
			$235.39

8.3 Bill of Materials and Budget

A break down of the materials bought and used in our project can be found in Table 2 below.

8.4 Risk Mitigation Plans

Since we have a robust classification algorithm, we are more worried about the object detection recognition and segmentation algorithms. To minimize this risk, we are using a white platform to make the segmentation a simple color thresholding problem. This will not only save us development time for the computer vision algorithm, but will also allow for a more robust algorithm, decreasing two risks for us.

Another risk that we have considered is the possibility of the back-end API suffering from degraded performance, bugs, or an outage altogether. Some of these possibilities, such as degraded performance and outages, may be caused by server availability and its network performance instead of the quality of our code or other factors under our control. To mitigate this risk, continuous uptime monitoring as described in section 7.2 were adopted, to provide us with real time alerts as these incidents occur. Similarly, the suite of tests run by the continuous uptime monitoring service also gives us more assurance on the correctness of the code. Bugs are also prevented from entering the master branch by enforcing strict type-checking and linting standards.

A similar source of bugs has been considered for the front end. Strict type checking and linting for the JavaScript on the front end will significantly decrease the number of bugs, as well as eliminate a large portion of time spent debugging. This will additionally increase the reliability of the system in production and during interaction as many hidden bugs can be removed before they cause issues.

9 RELATED WORK

9.1 SmolKat

SmolKat is a smart fridge system that can detect and track produce inside one's mini-fridge and display where the produce is on the shelf. It was a project completed by Team D3 in Spring 2021. However, unlike our outward-facing camera system, SmolKat used cameras mounted *inside* the mini-fridge. Their setup had the advantage of not disrupting their ordinary workflow of loading and unloading groceries out of the fridge, and even included a neat feature where an LED would light up to indicate the exact position of an item inside the fridge. These are features that we would be unable to achieve with our current system design. However, for reasons detailed in 5.3, this design suffers from some major flaws, including insulation problems with wiring, cost-integration concerns and occlusions. Indeed, in their final demo, we can see that their wiring was a complete mess, and would definitely have affected the insulation of the fridge as the door would not have been able to close properly. Moreover, their product seems to work only with mini-fridges consisting of a single layer, likely because they could not overcome the occlusion problem. On the other hand, our system will be an integrated, modular add-on that can be used for any normal fridge. It will be able to track a large variety and number of fresh produce, without being limited by fridge size.

9.2 Cozzo

Cozzo is a commercial fridge and pantry management application targeted at households who would like to track their groceries. Similar to our system, it offers reminders about expiring produce and suggests smart recipes to users. They offer many impressive features, with notable ones being the ability to scan receipts and barcodes of pantry produce. However, as noted in Section 5.1 above, the scanning of barcodes does not really apply to our use-case because most fresh produce (eg. fruits and vegetables) do not have barcodes and are handled manually at the store. That being said, their receipt scanning is a great idea, although most of the user input required is still fairly manual: Users still need to review each item on their scanned receipt individually, and need to manually update each time an item is used, changed or thrown away. On the contrary, our system allows users to scan produce before loading or unloading them into the fridge, which provides an intuitive way of tracking produce going in *and out* of the fridge; the ability to track produce being used is a unique feature of our design, and will allow us to also give the user a summary of their food (and possibly nutrition) consumption.

10 SUMMARY

FreshEyes aims to provide a modular, intuitive and low-latency experience to aid households in tracking their fresh groceries. To do this, we provide an integrated modular system that can be placed on *any* fridge, and that uses a

vision-based scanning system that is non-obstructive, accurate and fast, with an easy-to-use UI optimized for efficiency and intuitiveness. Our back-end is designed to be low-latency, secure and stable, thus providing our users with a smooth and enjoyable experience. Finally, we further add value to the user, by not only tracking their produce going in and out of the fridge, but also remind them of expiring produce via our web-based UI, and even suggest recipes to them that use said produce.

One major challenge will likely be the back-end subsystem, which has a lot of moving parts, and is also the backbone of our project. Getting the computer vision system to work reliably in a real-world environment will also be a challenge, and might require re-training the CNN with self-collected data - a time-consuming process. However, the most challenging part of the project is most likely going to be the integration of the various subsystems together through the back-end, which requires a well-defined APIs and the individual subsystems to be working properly. However, given our current abilities and the fact that we are on schedule implementation-wise, we are somewhat confident in the success of the project.

Glossary of Acronyms

Include an alphabetized list of acronyms if you have lots of these included in your document. Otherwise define the acronyms inline.

- API - Application programming interface
- UI – User Interface
- CV – Computer Vision
- CNN – Convolutional Neural Network
- ORM - Object-Relational Mapping
- BLE - Bluetooth low energy
- NFC - Near-field communication
- vCPU - Virtual Central Processing Unit
- RAM - Random access memory

References

[1] EPA, "Facts and figures about materials waste and recycling," 2018.

[2] USDA, "Fruit and vegetable prices," Aug 2019.

[3] E. A. Shrider, M. Kollar, F. Chen, and J. Semega, "Income and poverty in the united states: 2020," Oct 2021.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[6] H. Mureșan and M. Oltean, "Fruit recognition from images using deep learning," *Acta Universitatis Sapientiae, Informatica*, vol. 10, pp. 26–42, 06 2018.

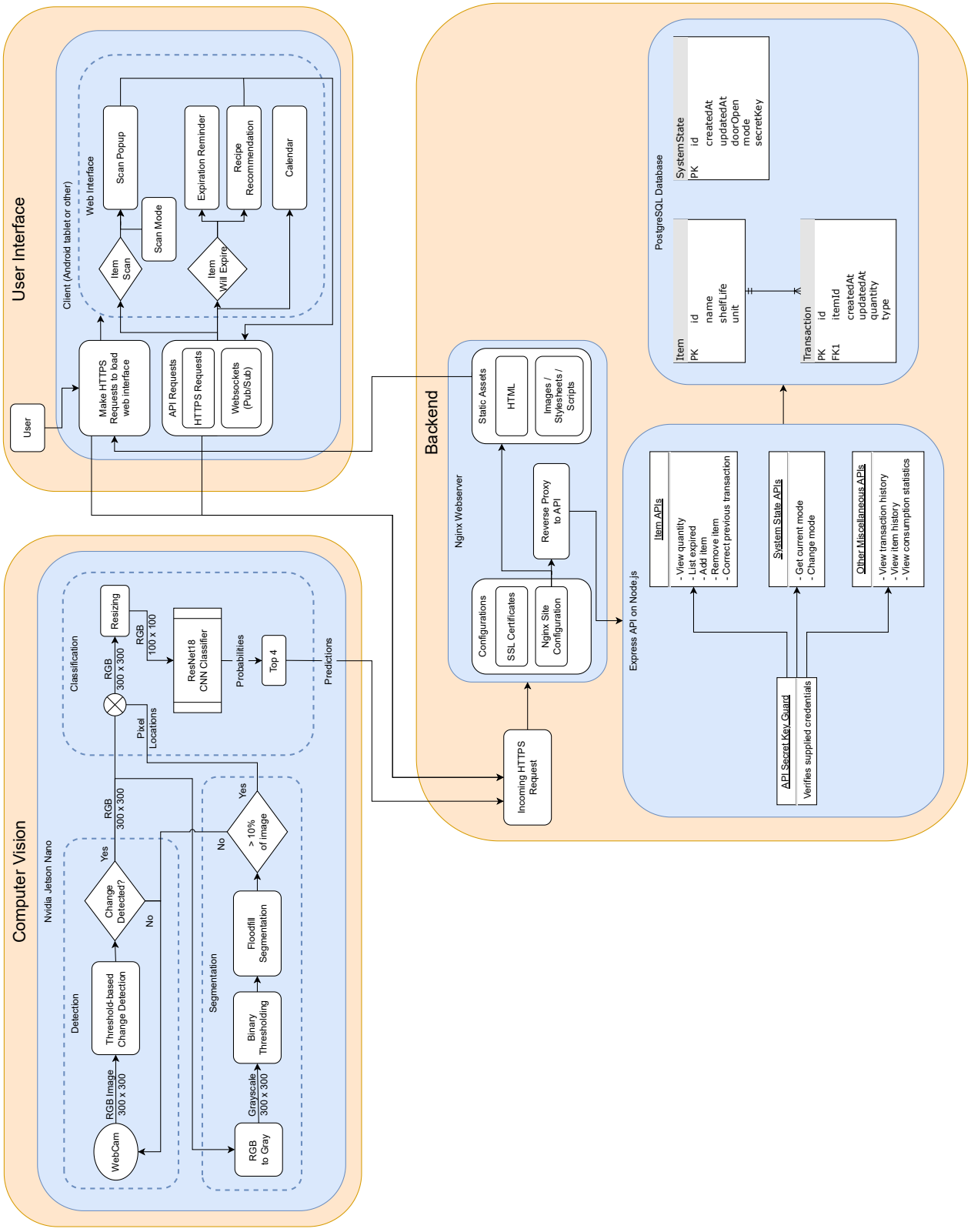


Figure 6: A full system block diagram of all components in our project

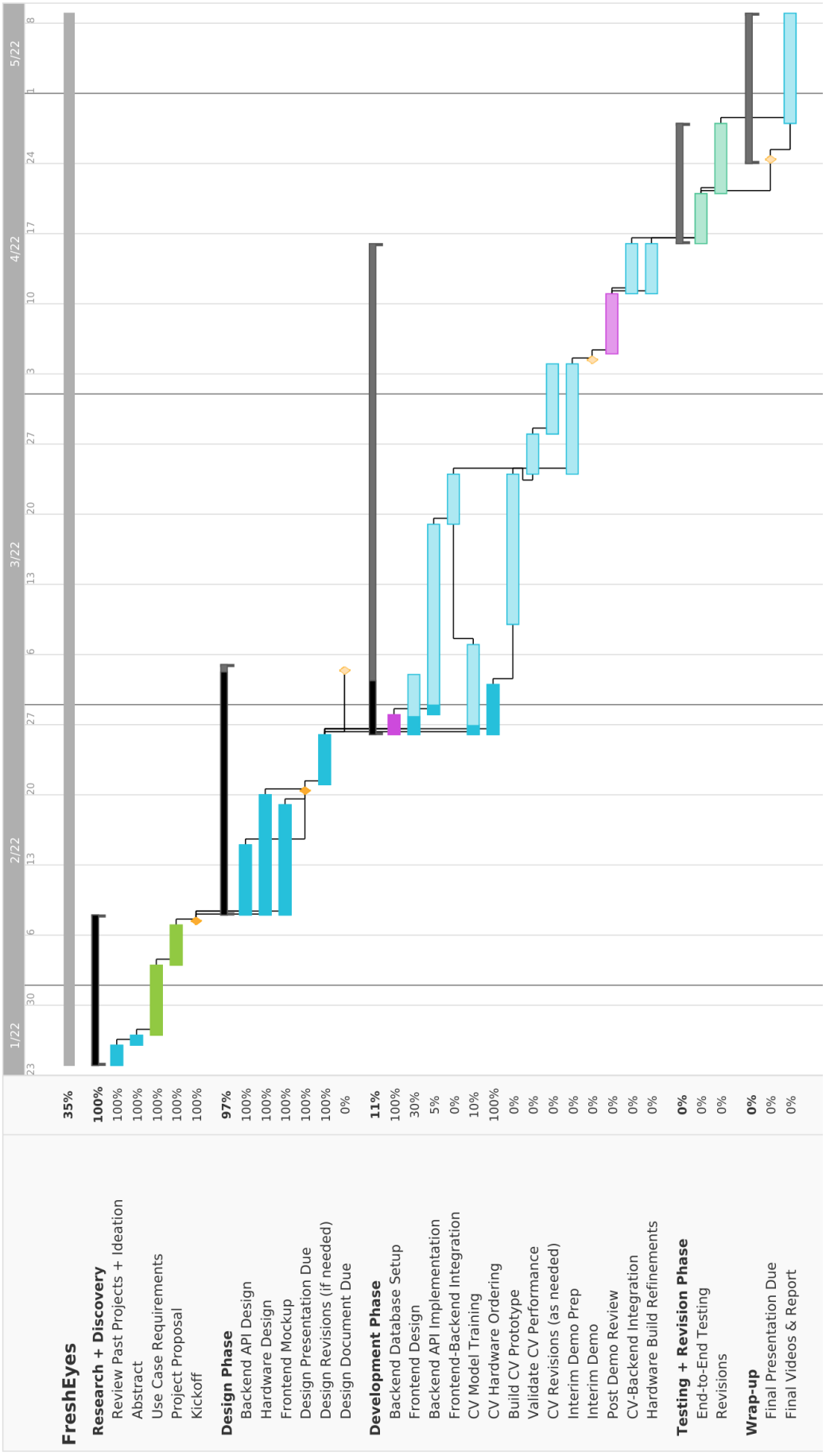


Figure 7: Gantt Chart and completion status as at March 4, 2022