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 will even suggest some recipes that utilize expiring foods.**

***Index Terms*—Computer Vision, Databases, Inventory Tracking, Mobile App, Neural Networks, 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 via in-app and email notifications, 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 vegetable a week. This would result in approximately $1.50 saved per week (the approximate cost of a produce item), and approximately 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!

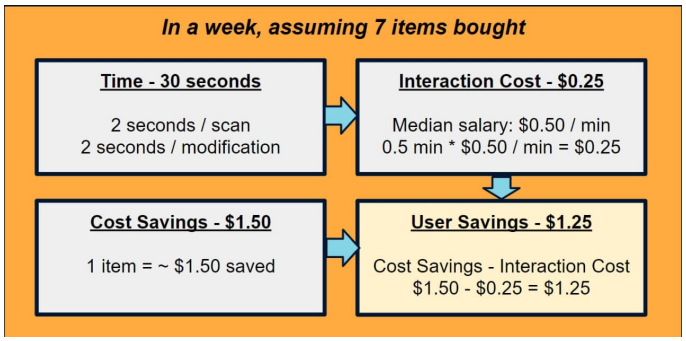


Figure 1: Visual Summary of Use-Case Justification

3 ARCHITECTURE AND PRINCIPLE OF OPERATION

Our architecture consists of 4 primary components or subsystems, whose relationships are briefly summarized in Figure 2 below, and in greater detail in Figure 10.

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

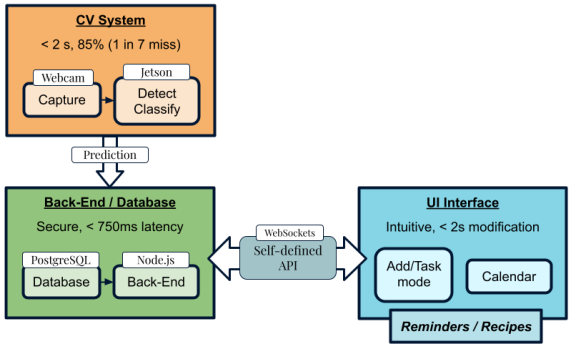


Figure 2: Summary of Architecture. Detailed diagram in Figure 10 below. Big rectangles represent key subsystems. Smaller, colored rounded rectangles represent key actions or components, with the uncolored rectangles above them being the implementations or hardware used.

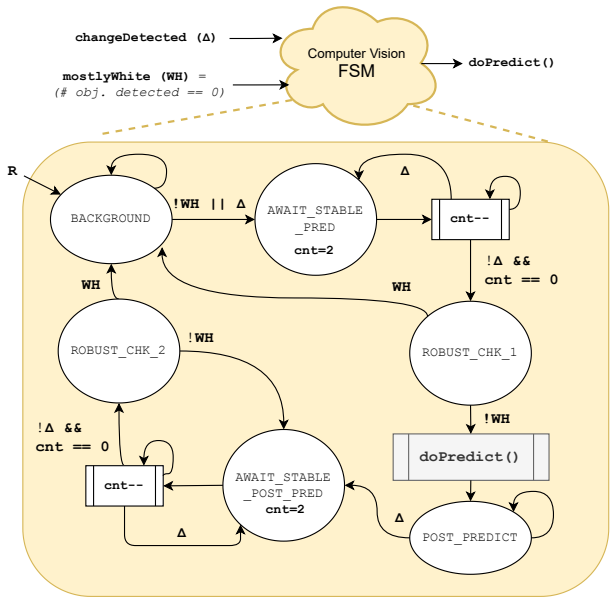


Figure 3: Detailed architectural diagram for Finite State Machine (FSM) of Computer Vision (CV) System

3.1 Computer Vision System (CV)

The computer vision (CV) system's main job is to capture, detect and classify the items presented to it by the user. Because we want a seamless scanning experience that minimizes button presses, our CV algorithm consists of a Finite-State Machine (FSM) that uses white-background and motion detection to detect the current state of operation (e.g. "waiting for item", "item on platform", "waiting for removal of item"). Notably, this FSM is what allows the user to simply place a fruit or vegetable on the platform and have the system automatically detect and predict its type — no button presses are needed to initiate prediction. Once the user removes fruit of the platform, the system goes back into `BACKGROUND` state and awaits the next item. Crucially, various robustness checks are in place to ensure that our FSM does not enter a limbo state because of unexpected behaviour. For example, even if the user removes items one by one from the platform (instead of all at once), the FSM will continue to stay in the `POST_PREDICT` state until all the items are removed from the platform. Similarly, the algorithm is robust against lighting changes and hand swipes (for example to play with the tablet). A detailed diagram of the FSM is shown in Figure 3 below.

The main workhorse of the CV system is the quantity detection and neural-network-based classifier, which is triggered once the FSM enters the `DO_PREDICT` state. Here, our algorithm will classify the image of the fruit and vegetable using the Convolutional Neural Network (CNN) outlined in Section 6.2 below. In parallel, the CV system performs quantity detection, where the number of items on the platform is detected using background thresholding, morphology operations and floodfill. The CNN will then spit out a set of top 5 predictions, which is then sent over, along with the detected quantity, 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 allows 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 foldable platform for the user to place their items for scanning. Using low-cost 3M Velcro Command Strips^{TM}, Our integrated board allows for the user to easily install (or uninstall) our system as a module on any fridge door — all in less than 5 minutes! 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. An annotated image of our *physical* integrated system is depicted in Figure 4 below.

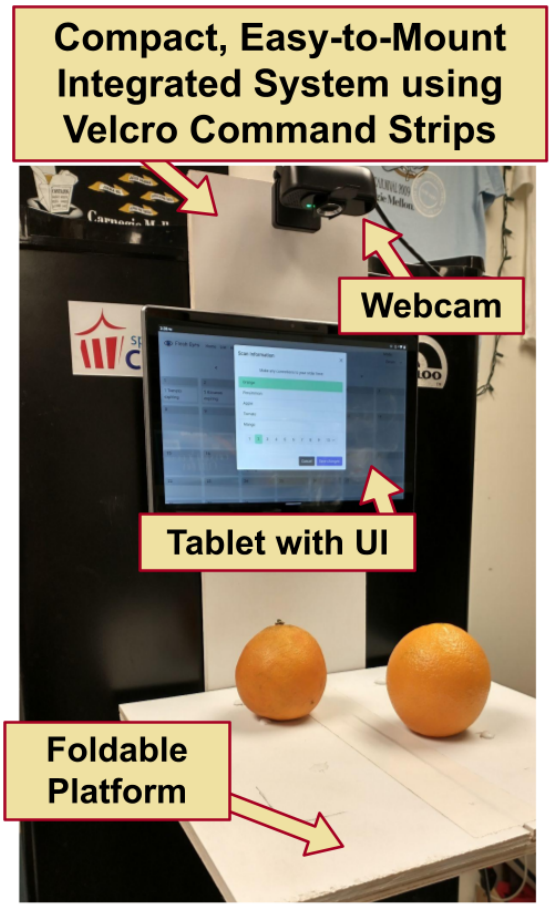


Figure 4: Annotated Picture of our Integrated Fridge-Attachment Module

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 100×100 RGB image as an input to the neural network, we need an RGB camera that has a minimum resolution of 400×400, 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.

Additionally, to facilitate ease-of-use, we would like our integrated system to be installed and uninstalled within half-an-hour (i.e. 30 mins).

As we present in Section 7 below, we comfortably meet our requirements, and even exceeded some of our expectations, such as the quick installation time.

5 DESIGN TRADE-OFF 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 4) 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 4 above, the integrated attachment system has a backing made out of a piece of $6'' \times 15''$ of solid plywood, onto which our tablet and webcam setup were mounted on. The plywood backing is designed to be installed onto any fridge door using simple 3M Velcro Command Strip^{TM} mounting tape.

Notably, our webcam setup involves a web camera pointing downwards towards a square platform that is made out of $12'' \times 12''$ of plywood, and spray painted white. 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.

The plywood backing and white platforms are screwed onto metallic folding shelf brackets which secures the two separate pieces together. We specifically chose *folding* shelf brackets because they allow the platform to be folded down when not in use, thus allowing for a non-obstrusive design.

6.2 Computer Vision System

The computer vision system first obtains a 800×600 image using the webcam video feed, taking frames every 200ms. This image is then center-cropped to a size of 450×450 to ensure a high-visibility square image that the classifier requires. Notably, as shown in Figure 4, the webcam is setup such that it is pointing down towards a white platform. Thus, we exploit this "whiteness" of the background state and use simple thresholding on the lightness channel of the Hue-Saturation-Lightness representation of the image in order to detect whether we are looking at the white platform, or a potential item to be scanned. This "white background check" (variable `WH`, see Figure 3 above) is crucial for the inner working of the FSM and is an important part of the robustness checking process.

Additionally, this thresholding is used for quantity detection, i.e. to detect how many objects there are on the platform. To do this, after thresholding, we perform morphology operations (erosion and dilation) on the binary image, followed by floodfill segmentation, to obtain the number of objects on the platform. An example of this is shown in Figure 5 below.

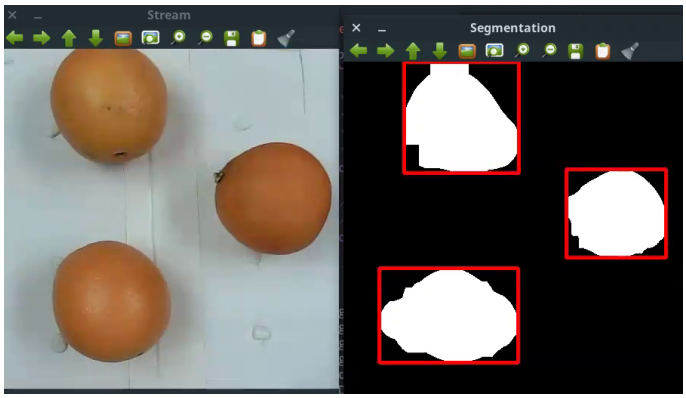


Figure 5: Visualization of Segmentation Process

However, checking for the "whiteness" of the input image alone is insufficient for robust detection of an item for the `DO_PREDICT` state. For example, consider the case of a hand moving into the camera's view to place an apple (or other item) down onto the platform; the camera is likely to capture the hand movement instead of the apple, and will result in inaccurate classification by the CNN. To overcome this issue, we enforce a "stability check" in the FSM (variable Δ), see Figure 3 above), where we detect motion, or lack thereof, by checking significant changes in pixels between consecutive frames. Using this motion detection, the FSM only triggers the `DO_PREDICT` state if it detects motion *and* the background is not white after stability (i.e. when no more motion is detected). This should only occur when the user places an item on the platform.

In the `DO_PREDICT` state, the captured image is resized to a 100×100 image and is sent through a ResNet50-based [4] convolutional neural network (CNN), which will output a set of probabilities. The top 5 probabilities and detected quantity are then sent to the back-end via a JSON request as part of our self-defined API, using the https://github.com/jpbarrette/curlpp`cURLpp` library. The information will subsequently be displayed to the user via the User Interface (Front-End) for confirmation (default) or correction (if any), thus allowing for an intelligent, semi-automatic tracking process. Notably, we trained the CNN ourselves using PyTorch [5] on a combination of the Fruits Recognition "Silver Tray" dataset [6] *and* our own self-collected data, which has a total of 16 classes of fruits and vegetables. They including common produce like apples, oranges, tomatoes and bell peppers. The data was collected under and augmented for different backgrounds, lighting conditions and rotations, which allowed the algorithm to very robust against different illumination conditions.

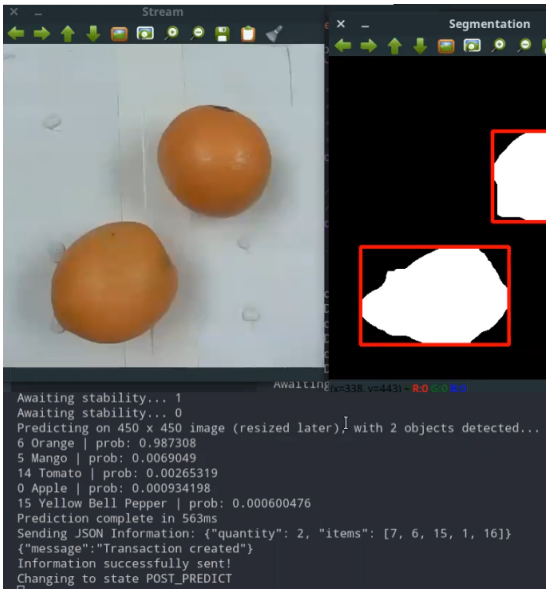


Figure 6: Algorithm in `DO_PREDICT` state. Notably, console shows the stability checks, top 5 probabilities, quantity detection and JSON request to our API.

After completing the scan/confirmation, the user will then remove the fruit from the platform, and in doing so, trigger another motion change in the CV FSM system. Similar to the predict state, the FSM will only return to the `BACKGROUND` state if a white background is detected. This is very important because the user could be removing items one at a time from the platform, and we do not want the FSM to return to the `BACKGROUND` state prematurely

and trigger a false detection.

For maximum efficiency, all image processing algorithms, including the neural network, are implemented in fast, optimized C++ code using the https://docs.opencv.org/OpenCV and https://pytorch.org/cppdocs/TorchLib libraries. The CV system also runs on the Jetson Nano, which is specialized for running CNNs.

This completes the verbal explanation of the entire scanning process, and how our CV system is able to robustly detect, and react accordingly to, every possible state of this process. Notably, the scanning process is very similar to checking produce out at a self-checkout kiosk at a grocery store, and is thus extremely intuitive to users. However, unlike many other systems, we do not rely on barcodes or RFID tags but instead use a universal vision-based classification system. Indeed, one might notice how seamless the entire process is for the user – they only need to perform confirmation via clicks on a conveniently positioned tablet (see Figure 4), and our algorithm takes care of the rest!

6.3 Database System (Back-End)

The core of the back-end was 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, as 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 are shown in the relevant back-end sections of the block diagram at Figure 10.

The database schema is defined as code (see Figure 7) in a JSON-like format which is ingested by an ORM, specifically Prisma, which creates and even updates the SQL tables based on the definition. This means that the schema and object relations are checked into and tracked by our version control system (Git), just like any other piece of code. Furthermore, using the ORM, we are also easily switching the underlying database system between SQLite on development instances and PostgreSQL on production systems, all while keeping the differences transparent to us as developers. 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 is deployed on Oliver's 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 minimum 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.

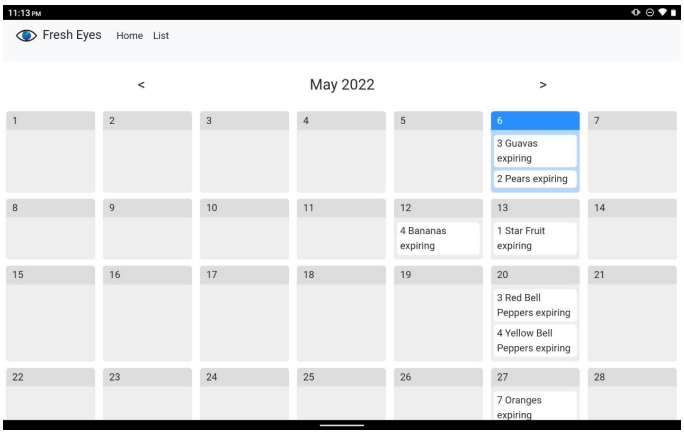
Figure 8 below shows the UI hosted on Alex's server (https://capstone.astrasser.com:2096/). Notably, we have designed our scanning interface (see Figure 8b) for maximal intuitiveness and minimal navigation. To this end, we show the top 5 most probable predictions as determined by the CV system, so that 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. Our Computer Vision System also helps to detect small quantities of items (about 2 to 4) to further reduce the number of clicks necessary. Additionally, we have designed the calendar view to be as intuitive and aesthetically pleasing as possible, shown in Figure 8a. In addition, users can easily view the list of items in their fridge from any Internet-enabled device, shown in Figure 8c below, by visiting the website and clicking the "List" button on the navigation bar. By clicking on the item in list mode, they are also able to view its nutritional information, shown in 8d below.

Our initial prototype involved a dropdown menu that made the user select whether they want to be in "Add", "Remove" or "Return" mode. However, user studies revealed that this was extremely unintuitive and was prone to error. Users often forgot to select the correct mode before a scan, and were unable to change the mode they were in after the scan. To solve this problem, we changed from the dropdown menu into a set of 3 buttons, corresponding to each respective mode, on the scan confirmation dialog instead, shown in Figure 8b. Subsequent user studies showed a 100% increased preference for this design.

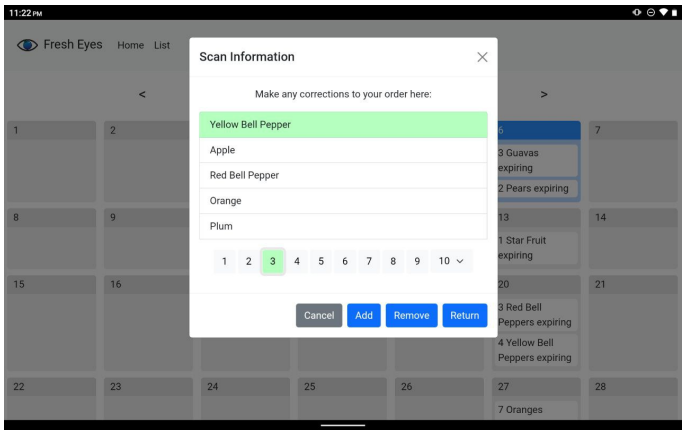
```
model Item {
  id           Int                    @id @default(autoincrement())
  name         String
  shelfLife    Int // Number of days item can be stored
  unit         String
  transactions TransactionsOnItems[]
}

model Transaction {
  id         Int                  @id @default(autoincrement())
  createdAt  DateTime             @default(now())
  updatedAt  DateTime             @updatedAt
  items      TransactionsOnItems[]
  quantity   Int
  type       Int // 0 = addition to fridge from store, 1 = removal, 2 = return to fridge
}
```

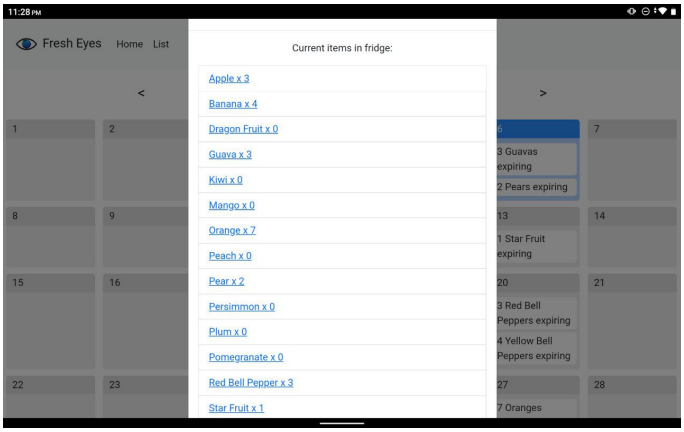
Figure 7: Sample of database schema defined in code, with relations explicitly defined



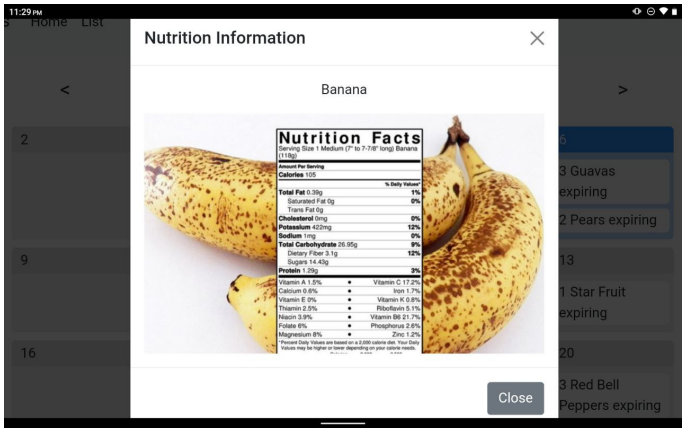
(a) Calendar View



(b) Scan Interface



(c) List View



(d) Nutritional information displayed after clicking “Banana” in Figure 8c

Figure 8: User Interface (Front-End) as displayed on tablet

7 TEST & VALIDATION

7.1 Results for CV Design Specification

We ran a couple of tests for the speed and accuracy of the computer vision design specification. First, we tested the accuracy of the CV system by running 20 scans among randomized fruit selection in varied lighting conditions. Of the 20 tests, 18 of the fruits were detected correctly and were the first recommendation, 1 fruit was the second recommendation, and 1 fruit was the third recommendation. This means that all 20 fruits came up successfully on the display, and we got 90% fully successful scan rate, greater than our 85% target in difficult conditions.

Additionally, we tested the scan speed in time from fruit placement to UI prompt. This used the same random fruit selection and randomized lighting as the previous tests. On an average over 10 trials, we got an 1.67 seconds in time, beating our target 2 seconds by a third of a second.

7.2 Back-end Testing Results

For the back-end, we wrote a suite of automated tests on Postman, an API client and development environment. Some of the tests covered the response correctness of the back-end, e.g. ensuring that creating and confirming a new addition transaction resulted in an increase in the number of fruits of that type. Using Postman, we could also monitor the response latency, ensuring it fell within the bounds of the design specification.

These tests and checks were then deployed onto the cloud, where they ran automatically on a set schedule. This strategy ensured constant monitoring of the back-end, as well as timely alerts if any code deployments or updates broke the correctness of the back-end, or significantly affected the response latency. This allowed us to quickly isolate any breaking changes, identify the offending code and fixing the bug before the problem snowballs into something that required significant amounts of effort to troubleshoot and debug.

Naturally, we kept the back-end such that all API endpoints passed all the correctness tests. As for the latency, the p99 value measured once every 15 minutes over 24 hours was 246ms. In other words, within this timeframe, 99% of the automated test API requests had a round-trip time of 246ms, significantly lower than the latency of 750ms specified in our use case requirements.

7.3 User Studies for Front-end

We tested the user interface prompt with 3 individuals over 5 tests each to get the user response time for the popup. Our design specifications placed this at less than 2 seconds. Despite some some variability in response time, we achieved an average of 1.52 seconds per prompt, beating our target by nearly half a second. In addition, we also collected qualitative comments about the experience from our participants, which resulted in some important improvements to the interface, such as the change from dropdown menus to buttons for mode selection (see Section 6.4), and the inclusion of email notifications (see Section 6.3).

7.4 Integrated Pipeline Robustness Test

Because we emphasize a smooth, seamless and robust user experience, we put our entire pipeline through multiple stress tests. To this end, we kept adding, returning and removing items from the fridge, by an arbitrary number of times and in arbitrary order, and intentionally including a non-exhaustive list of various edge cases which could potentially break the system:

Table 1: Examples of test cases for system robustness

Test	**Testing for...**
Turning lights on and off	Different lighting conditions
Swipes across platform	False detections
Consecutive large amounts of single item	Overflow errors
Remove more items than are present	Negative quantity
Add/remove multiple items one by one	CV robustness check

This extensive stress test allowed us to find bugs and make our system more robust.

8 PROJECT MANAGEMENT

8.1 Schedule

The project's Gantt chart as at May 6, 2022, is shown in Fig. 11. All tasks prior to the final report has been fully completed on schedule. Our project did not actually require much changes to the schedule, as slack and leeway was built into the schedule at every milestone. For example, in the design phase, 1 week of slack was built into the schedule to allow for any needed design revisions. As none was needed, this actually resulted in an additional week of development time for the development phase. However, some of the slack time in other phases were indeed utilized. The development phase also had 1 week of slack time for any needed CV revisions. As can be seen from the Gantt chart, this time was then allocated for further dataset collection and training in order to build an even more robust model. Thanks to good planning in the initial phases of the project, no changes were needed to the schedule in any phase of the project, with the slack time more than covering for any unanticipated work required - in other words,

the known unknowns were well accounted for from the very beginning.

The major milestones for the project were:

- Research + Discovery: Ideation, coming up with an use case, and defining the goals of the project
- Design: Defining the architecture, subsystems, and APIs used in the project, building front-end mock-ups, culminating in a design document
- Development: Implementing the design, training the CV model, building an MVP, and further enhancements beyond the MVP stage
- Testing + Revision: Performing end-to-end testing, bug fixing, and doing any needed revisions
- Wrap-Up: Final presentations, videos, demo, etc

8.2 Team Member Responsibilities

Each member contributed equally to the project, specializing in a specific subsystem that corresponded with the individual's prior experience. Each member also occasionally helped with another member's part. These responsibilities are summarized in Table 2 below.

Table 2: Summary of Team Member Responsibilities

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

8.3 Bill of Materials and Budget

A break down of the materials bought and used in our project can be found in Table 3 below.

8.4 Server Usage

For this project, we had the benefit of members who had self-hosted servers and did not have to rely on services such as AWS for paid hosting. We were able to host the UI interface on Alex's server and the API on Oliver's server. We were additionally able to train the neural network on Alex's server. In a professional deployment, the server hosting for the web interface could be either condensed onto the API server, or offloaded to a free static site host/CDN such as GitHub Pages.

To further decentralize, the API could run locally on the Nvidia Jetson. This solves several issues for both us and consumers but comes at a price. The benefits include no reliance on external internet connection for the consumer (though this would still be necessary if external access was wanted), user control over their own data, and decreased server hosting cost for the producers of the product. However, in decreasing the centralization, it makes it much more difficult to push security patches and add features. Due to these trade offs, development was much easier to do in a centralized manner, and further research and development is necessary to determine whether a centralized or decentralized approach is more beneficial for our product.

8.5 Risk Management

This project, like all other projects, was not without its risks. The largest risk at the beginning of the project is the performance of the CV classifier, specifically the risk that the classifier is not robust or accurate. To mitigate this risk during our project, we were selective with the type of fruits we trained our model on. We also had backups of our models in case we wanted to revert to them, or perform transfer learning on them. Occasionally, after the model was trained on a new type of fruits, the overall accuracy of the model fell drastically. In these cases, we made judgement calls, by leaving out fruits that easily confused the classifier (e.g. orange vs grapefruit) and selecting only the most common types of fruits to be included in the model, ensuring that all common fruits are still covered in the process.

After we had a robust classification algorithm, the largest risk then involved the object detection recognition and segmentation algorithms. To minimize this risk, we used a white platform to make the segmentation a simple color thresholding problem. This not only saved us development time for the computer vision algorithm, but also allowed for a more robust algorithm, mitigating two risks for us at the same time.

Another risk that we had to address 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 altogether by enforcing strict type-checking and linting standards.

A similar approach with type checking (TypeScript) and linting for the front-end has also significantly decreased the number of bugs, as well as eliminating a large portion of time spent debugging. Since both the front-end and back-end were written in TypeScript, type definitions for data moving between these two systems (e.g. WebSockets) were shared, ensuring that both systems were sharing the same type definition, preventing bugs from "mis-typing" from occurring. This is an additional way of protecting the reliability of the system in production and during interaction as many hidden bugs can be prevented from being pushed to master altogether, eliminating them before they have the opportunity to cause any issues.

One other important risk mitigation factor adopted at

Table 3: 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
Wood	Craft Wood	Amazon	$10.99
Paint	White spray paint	Rust-Oleum	$13.47
			$235.39

the beginning of the project was adopting a version-control system that would allow us to revert to an older version or retrieve code in the event of a local data loss. In particular, all parts of our project were constantly synced on a private GitHub repository, and we made use of `git` tags to mark major milestones.

9 ETHICAL ISSUES

By design, FreshEyes is meant to target an ethical issue of food waste, aiming to reduce food waste at the slight cost of user inconvenience (i.e. constantly having to scan items in and out of the fridge) and perhaps electricity usage (since powering the tablet and Jetson still requires a low electricity cost). However, we believe that the benefits of our system outweighs the cost for several reasons. Firstly, our design aims to minimize user inconvenience by making the scanning process seamless, thus reducing cost of user inconvenience significantly. Secondly, the tablet and Jetson ($\approx$ 0.03kWh [7]) draws negligible power compared to the fridge that is being used ($\approx$ 1.5kWh [8]). Thirdly, food waste is a globally growing problem [1], and our product is estimated to help save approximately 25 lbs of food waste per year, per person (see Section 2 above). Therefore, we believe that our product's benefits greatly outweigh the cost in ethically combating food wastage.

Most of the other major ethical issues we considered were related to security, in particular to what happens if our databases are hacked and information is leaked. To this end, our databases were intentionally built with security in mind, with features such as authentication using API keys, with details in Section 6.3 above. Even in the worst case scenario with a complete data breach, as no personal information is being stored, the only data that an attacker is able to obtain is the list of items in one's fridge and what kinds of food the person likes to buy. This is not inherently sensitive information, and cannot be used to identify anyone. Moreover, since all image processing is localized on the Jetson by design (i.e. not sent over the web), no images of the user's face or house can possibly be leaked without physical access to the hardware.

However, during the ethical discussion forums, it was raised that serious ethical problems would actually come from the commercialization of our product rather than the security end. This concern is valid since our product was designed and intended for commercial use. For example, suppose that FreshEyes hits the market and becomes popular. Then, fruit companies like Driscoll or Del Monte are likely to offer huge amounts of money for targeted advertising to, say, users of our system that like buying strawberries and bananas. We would then be effectively "selling out" our users' data to third parties without their consent, which is unethical. Even if we did obtain their consent through "Terms and Conditions", we still believe that in doing so, we would be deviating from our initial goal of reducing food waste, and instead unethically re-intending our product as a money-making machine. Of course, the large amount of money will still be tempting, and will be hard to turn it down. Even if we resolve against explicitly selling away users' data, we might implicitly do so by selling our product to a company that might want to use it for said purpose. We believe that the best way to address this issue is to enforce strong data protection policies within the company, and open-sourcing all the code, including the backend. This way, anyone can host the service completely independently, thereby serving as a "poison pill" against capture by such corporate interests.

We start entering gray areas if the data is instead used for a "good cause". For example, health organizations and the government might become interested in our users' food consumption patterns to aid the creation of better programs and interventions that improve the population's overall health and nutrition. In this case, we are still deviating from our initial sole purpose of reducing food waste, but are also potentially contributing to healthy lifestyles for entire populations. In this case, it is probably justifiable to sell or "donate" the users' data for positive and ethical research purposes; however, this must notably be done with users' explicit consent.

10 RELATED WORK

10.1 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 *in-*

side 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.

10.2 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.

11 SUMMARY

FreshEyes aims to provide a modular, intuitive and low-latency experience to aid individuals and households in tracking their fresh groceries. To do this, we provide an integrated modular system that can be easily mounted on *any* fridge within 5 minutes, and that uses a vision-based scanning system that is non-obstructive, accurate and fast. It also boasts an easy-to-use UI optimized for efficiency and intuitiveness that is accessible from any Internet-connected device. 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 email notifications, and providing them with nutritional information of the fruits and vegetables they have consumed.

11.1 Future work

Currently, our system supports only one user. Ideally, we want multiple users, where each user has an individual inventory. In and of itself, implementing this feature would not be difficult due to the inherent scalability of the ORM Schema on the back-end. However, we were approaching the tail-end of our project timeline with a fully-working system that demonstrated all the important features well; thus, to mitigate risk, we decided that the potential bugs associated with adding this feature without adequate time for testing was not worth it. That being said, if given more time we would definitely add, and robustly test, this feature of multiple user accounts.

Our CV system assumes that all items placed on the platform are of the same single type, and is trained for such. For example, the CV algorithm accepts 3 oranges, but not 1 apple and 1 orange. This is inefficient as the user could have otherwise scanned, say, 4 types of fruit at once. However, we noticed something interesting about our classification algorithm, as shown in Figure 9 below. Despite bring trained to only classify one type of fruit/vegetable in a given image, it was able to predict 3 different classes of fruit (in this case, apple, banana and starfruit) as the top 3 probabilities.

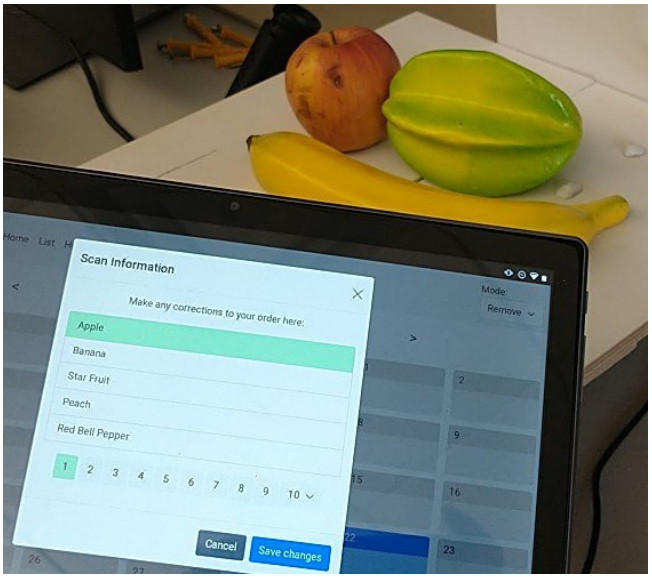


Figure 9: CV Classifier predicting 3 different classes as the top 3 probabilities despite being trained to output only 1 class at a time.

This experiment was repeated with different combinations of fruits, and we obtained mostly the same results, sometimes with one of the items being 4^{th} or 5^{th} on the list, but the top 5 would always contain every item.

One of the solutions we thought of implementing was making the top 5 predictions on the UI being checkboxes (multiple selections) instead of radio buttons (single select). However, our algorithm would then have problems trying to associate quantity with the individually-selected classes. One workaround is to have the row of quantity selections to be different for each item, which would require a major overhaul of the front-end. However, if given enough time to implement and test, it is definitely a feasible solution, and would greatly improve the user experience.

One major roadblock that we faced in the project was with setting up the project on the Jetson Nano. While the Jetson is *theoretically* optimized for running machine learning code, Samuel's personal computer was able to run a single classification in and average of 35 ms compared to the Jetson's 150 ms. This is a 5 $\times$ reduction in latency, and was barely acceptable for our design requirements. In addition, we realized that the "initialization" (i.e. first prediction) always took a full 3 minutes to run on the Jetson. The problem came down to not having enough RAM (only 2 GB), resulting in the Jetson Nano having to move data between RAM and swap when loading our model and performing the first prediction. However, once the Jetson "learned" how to juggle between RAM and swap, future predictions wereo h much faster, albeit still at a slower 150ms compared to that of 35 ms on Samuel's laptop boasting 16 GB RAM. Thus, for future work, we would want to get a beefier hardware system, such as the Jetson Xavier which has much larger RAM (4GB vs 8GB), memory (2MB vs 6MB L2 cache size) and GPU (128-core Maxwell vs 384 CUDA cores) capacities.

11.2 Lessons Learned

We were able to accomplish most of what we had envisioned at the start of our project, since we had some general experience in each sub-area of our project. This allowed us to correctly allocate and schedule time at the beginning. However, we wish there was more time to polish the project even further through improving the user experience, since it can feel a little unintuitive at times with the platform, scanning, and placing in the fridge. Overall, we were able to learn a lot about developing real-world, production, user-facing applications and were able to produce a complete, useful product.

As for specific implementation lessons, we learned that it's a lot easier to get better equipment for quicker development, and then make performance improvements to make your code run better. The Jetson Nano was the right tool in the long run, but getting the code to run took many extra hours of work just to get something both working in general AND working on the Jetson, instead of working on getting something working roughly first.

Additionally, it was incredibly important to start early. We were able to experiment with the machine learning a lot and tune it to work well for this project. We ended up completely scrapping our original dataset and machine learning model, and although retraining took a decent amount of time, it wasn't an issue since we had started with plenty of time to spare and stuck to our schedule.

Glossary of Acronyms

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

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] I. Hussain, Q. He, Z. Chen, and W. Xie, "Fruit recognition dataset," 07 2018.

[7] "Guide to technology power consumption: Compare the market," 2020.

[8] R. McCarthy, "How much power a fridge uses - in watts, cost and kwh," Nov 2019.

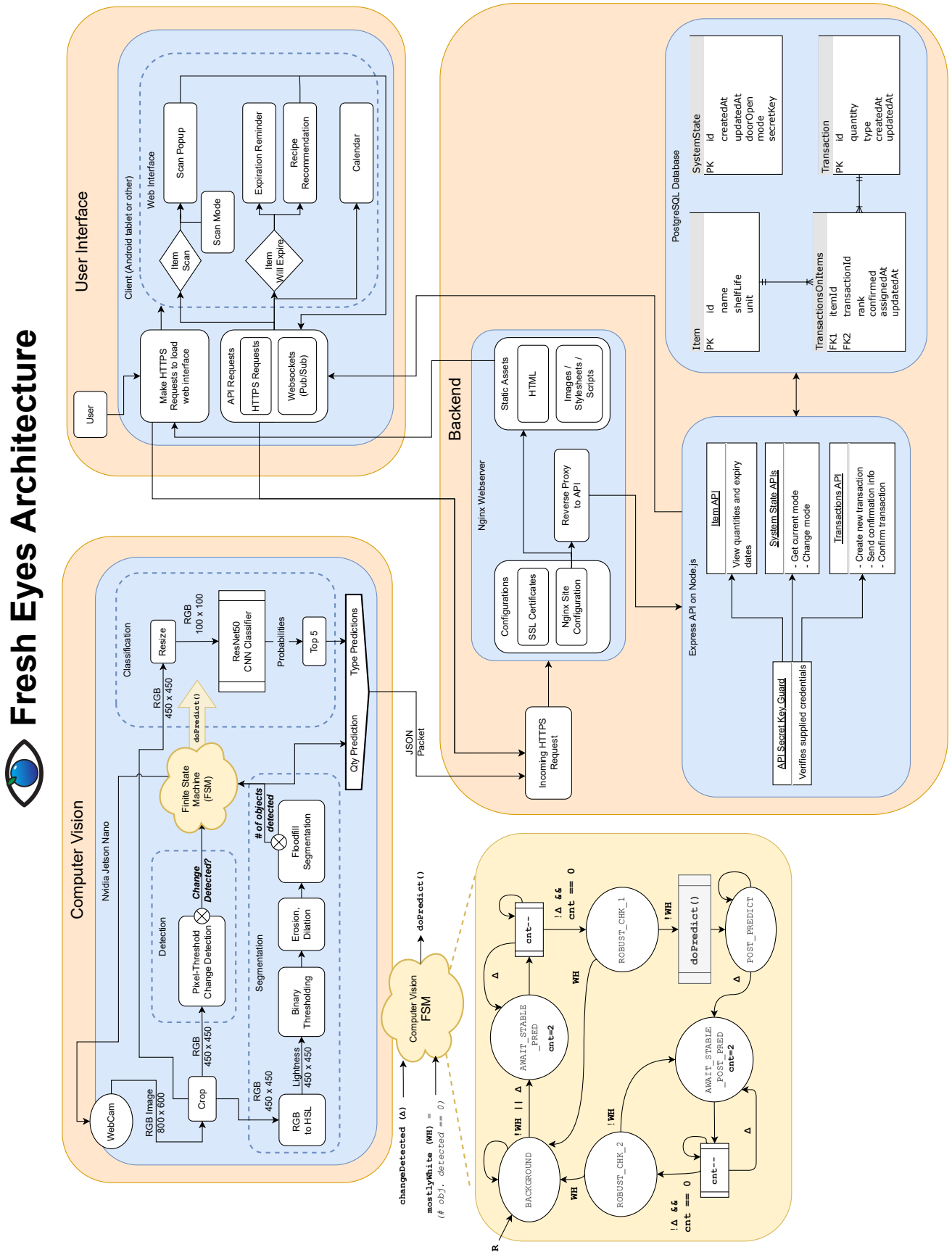


Figure 10: A high-resolution full-page version of the our system block diagram explained in Section 3

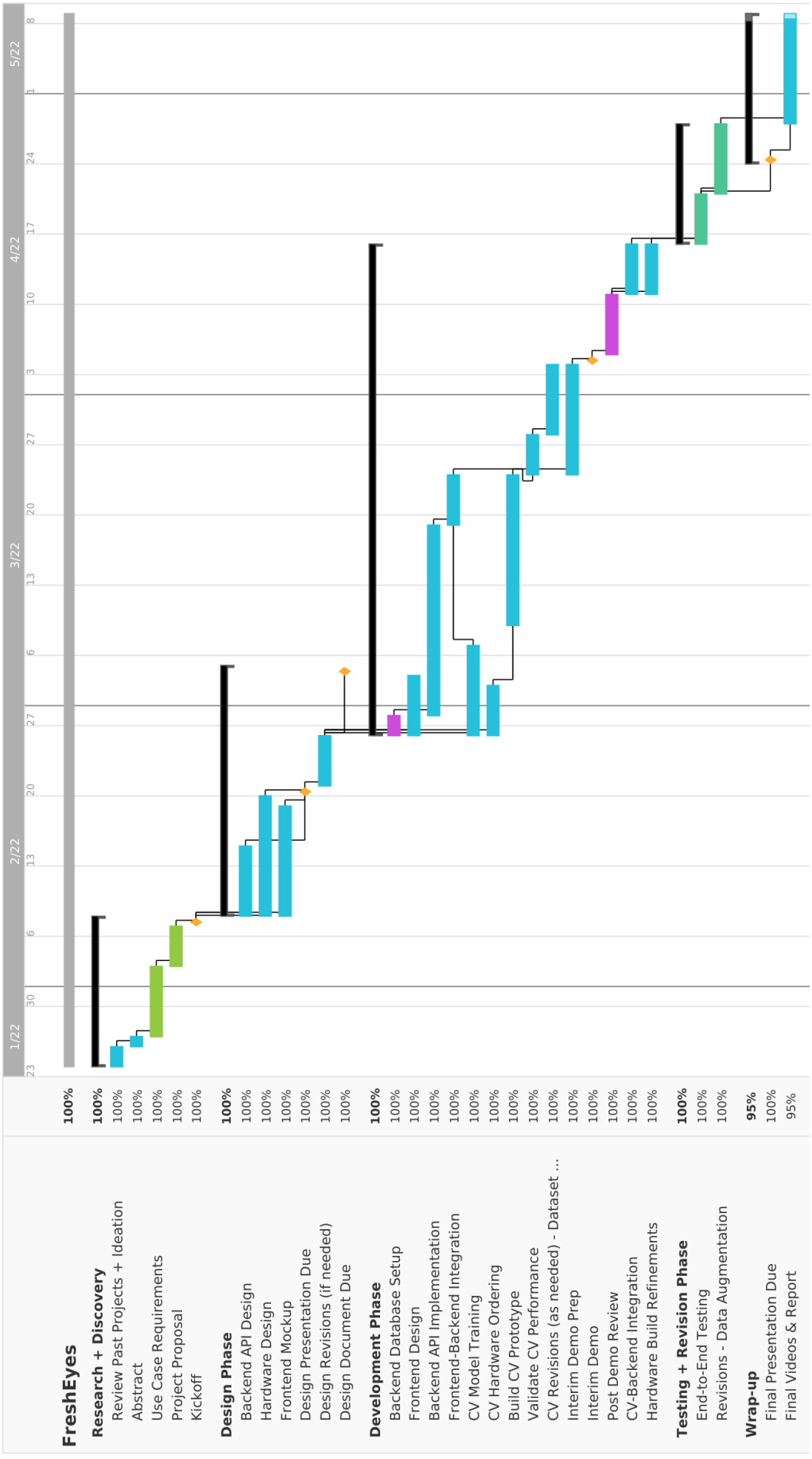


Figure 11: Current project Gantt chart as at May 6, 2022