6-14-2023

A Hardware Platform for Wireless Beehive Monitoring

Erik Wrysinski

Jonathan Stock

Collin Paiz

Daniel Blanc

Santa Clara University

Department of Computer Science and Engineering

Date: June 14, 2023

I HEREBY RECOMMEND THAT THE THESIS PREPARED

UNDER MY SUPERVISION BY

Erik Wrysinski, Jonathan Stock, Collin Paiz, and Daniel Blanc

ENTITLED

A Hardware Platform for Wireless Beehive Monitoring

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

DocuSigned by:
Behnam Dezfouli
6ED175D6B91A4FF...
Thesis Advisor
Dr. Behnam Dezfouli

N. Ling
N. Ling (Jun 16, 2023 12:07 PDT)
Chairman of Department
Dr. Nam Ling

A Hardware Platform for Wireless Beehive Monitoring

By

Erik Wrysinski, Jonathan Stock, Collin Paiz, and Daniel Blanc

Submitted in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science
in Computer Science and Engineering
in the School of Engineering at
Santa Clara University,

June 14, 2023

Santa Clara, California

Acknowledgments

We would like to thank Gerhard and Lisa Eschelbeck, Wendy Mather, and Kian Nikzad from the California Master Beekeeper Program for the project idea and their continued support and advising throughout the project.

We would also like to thank our advisor Dr. Behnam Dezfouli for his help throughout the past year on the project by providing insight whenever we would get stuck. He has also been of great help when modifying content for our thesis.

Finally, we want to thank Cheng Zhang, Jason Fong, Tim Lu, Chan Nam Tieu, and Niyibitanga Inosa for their contributions to our project through the development of the machine learning model.

A Hardware Platform For Wireless Beehive Monitoring

Erik Wrysinski, Jonathan Stock, Collin Paiz, and Daniel Blanc

Department of Computer Science and Engineering
Santa Clara University
Santa Clara, California

June 14, 2023

ABSTRACT

Traditional beehive monitoring systems suffer from many challenges. These monitoring devices are expensive to set up, difficult to implement, and lack cross compatibility with each other. Preexisting beehive monitoring solutions face all of these shortcomings. Our beehive monitoring platform aims to overcome these issues by using inexpensive, off-the-shelf, open-source hardware paired with a computer vision machine learning model to accurately monitor the ingress and egress of bees into and out of the hive. This data is presented to the beekeeper in a simple GUI which allows them to track hive activity over time, and by extension, the overall health of the hive. All of the code in this project is open-source while still maintaining a professional look. This enables users to customize it to their needs. However, even if the user has no prior coding experience the proposed solution is easy to setup and run. The final product should alleviate many challenges that other beehive monitoring systems face and should hopefully create a disruption in the beehive monitoring market that would inspire other companies to utilize more cost efficient hardware and open-source software.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

API Application Programming Interface

AWS Amazon Web Services

CCD Colony Collapse Disorder

EPA Environmental Protection Agency

GUI Graphical User Interface

OBS Open Broadcaster Software

OS Operating System

SCU Santa Clara University

UI User Interface

Chapter 1
Introduction

1.1 Motivation

Bees (and other pollinators) play a fundamental role in the the ecosystem by facilitating pollination, which is essential for the reproduction and survival of many plant species. Bees pollinate plants allowing them to breed, thus producing fruit or seeds [1]. About 75 percent of our crop production is dependent on bees and other pollinator insects [2]. Without the help of bees, massive agricultural operations and natural processes that feed the Earth as a whole would cease to function. In recent years, bee populations have been on the decline due to external factors such as climate change and the use of pesticides on crops. These harmful influences have resulted in widespread Colony Collapse Disorder (CCD) [3]. CCD is a "phenomenon that occurs when the majority of worker bees in a colony disappear and leave behind a queen, plenty of food and a few nurse bees to care for the remaining immature bees and the queen" [3]. CCD can result in catastrophic beehive failure, but beehive monitoring systems can help identify this downward population trend prior to a beehive's collapse. Beekeepers serve a very important role in this respect, acting as caretakers for the bees and the entire hive(s) altogether. Monitoring a beehive's health, is an exceptional way to support bees' functions. One of the principal metrics that is used to monitor the health of a hive is the amount of bee activity the hive experiences [4]. This is something that can be easily observed in the field, but it can be difficult to accurately track remotely; even with a remote camera, you need active monitoring to track the

number of bees entering and exiting the hive. Providing a solution to this problem would not only make the lives of beekeepers easier, but it would also provide additional data in order to help the beekeepers have an accurate measure of a hive's health over time.

1.2 Problem

There is already a wide variety of existing solutions to this problem, ranging from *Eyesonhives's* camera system [5], to *Arnia's* hive scale [6], to *Broodminder's* temperature monitor [7], each of which measure different metrics in order to determine a hive's activity and health.

Although these pre-existing solutions work well, they each have a number of shortcomings. The biggest concern which we are working to address is the cost of these devices. Even a simple hive monitoring camera has a price point of over $350, and for more advanced systems such as *Arnia's* scale ecosystem, prices can soar over $900. These prices only include a single device, and for most beekeepers with numerous hives, these prices put these useful technologies out of reach. Another drawback to the current solutions are their closed-source ecosystems; beekeepers must purchase all components of the ecosystem from the same company, indefinitely locking them into that company's products/services.

1.3 Solution

While there are already multiple devices on the market which solve the problem we are addressing, they fall short in the categories of affordability and open-sourced ecosystems. The novelty of this solution is its low-cost and availability. We have built this device using entirely off the shelf, easily-accessible parts such as the Raspberry Pi and affordable image sensors. This will allow more beekeepers to acquire and use this valuable technology to

help track their hives' health, as well as allow them to modify or repair it themselves at a much lower cost than other solutions.

This system is easy to install and simple to use. The device can be easily deployed to monitor a hive's activity without needing outside input. This solution also uses a computer vision model provided by a separate Senior Design group to track the number of bees coming in and out of the hive. This data is viewable in a web dashboard, allowing for easy monitoring of hive activity and viewing of aggregate data collected by the device. This will allow beekeepers to more affordably track their hive activity, resulting in better care for the bees and their hives. This data is very valuable to the beekeepers, as the hive activity can be a direct indicator of hive strength. Additionally, the machine learning model can be trained to recognize swarming as well as training flights from young, newly emerged bees, which will allow the beekeepers to have a much better understanding of their apiary.

Chapter 2
Related Work

In this chapter, we will discuss related work by highlighting some of the preexisting solutions to beehive monitoring.

2.1 Arnia

Arnia [6] is a hive-monitoring system designed around the use of digital scales and an internal multi-purpose sensor which measures temperature, humidity, light, sound, and movement within the hive. *Arnia* is one of the most advanced hive-monitoring systems on the market, with a significant amount of data being captured for each hive. However, this comes at a cost. At minimum, each hive requires an investment of over $700, which includes a scale, gateway, and internal hive-sensor. In order to track an entire apiary using *Arnia's* equipment, a beekeeper is expected to invest thousands of dollars into their tech-ecosystem. Additionally, *Arnia's* systems require a yearly subscription to a cellular data plan in order to send their data back to *Arnia's* servers, which tacks on an additional annual cost of $150 per hive gateway (one gateway supports up to eight individual sensors). This is another major expense that a beekeeper would need to worry about. The last problem *Arnia's* ecosystem faces is its closed-source nature. This system is entirely powered by *Arnia's* proprietary servers. This prevents beekeepers from utilizing any of their own data storage solutions, and forces them to continue to pay to use the *Arnia* ecosystem.

2.2 BroodMinder

BroodMinder [7] is a scalable hive-monitoring system that uses temperature, humidity sensors and scales to oversee a beehive. *BroodMinder's* product is simpler than *Arnia's* and is cheaper as a result. This system is designed to be used in any configuration, so beekeepers can choose to use as many, or as little sensors as they would like. However, despite the simplicity, it is still a large investment for the beekeeper. Regardless of how many sensors a beekeeper uses, they are still required to have a Wi-Fi hub, which costs $220. On top of this, a single temperature sensor can cost over $65, and a scale can cost over $200. More advanced data collection (sound and activity) is available, but costs another $200 on top of the other costs. This puts the price of a *BroodMinder* system out of reach for those with a large number of beehives. Additionally, *BroodMinder* faces the same closed-source drawback as *Arnia*. Powered entirely by *BroodMinder's* servers, beekeepers have no choice but to continue to pay for the *BroodMinder* system.

2.3 Eyesonhives

Eyesonhives [5] is a hive-monitoring system that shares many similarities to the device that we are developing. It uses a camera pointed at the entrance of a hive, alongside machine vision analytics to determine hive activity. One major difference between *Eyesonhives's* implementation and ours is the camera location. In *Eyesonhives's* system, the camera is pointed towards the front of the hive from a distance, rather than from the side with close proximity to the hive's entrance. This system also faces similar problems as the ecosystems previously mentioned. A single camera is $380, with an additional cost for the analytics service, which is necessary for the system's continued functionality. However, this cost is not mentioned on their website, as each camera comes with one year

of free service. This may result in some beekeepers investing in *Eyesonhives's* ecosystem, only to find out a year down the line that they need to continue paying a substantial amount to continue using *Eyesonhives's* monitoring software. This system also struggles with the same closed-source issues as discussed with the previous solutions.

Chapter 3

System Architecture

After highlighting the drawbacks of other beehive monitoring systems, we can now look at how this solution aims to solve these problems. This chapter serves as an overview of this project's fundamental system design.

3.1 Web Application's Dashboard

The *Dashboard User Interface* utilizes a wide variety of industry-standard technology to deliver a state-of-the-art user experience. The web application dashboard is built using the *MERN* stack, a popular full-stack software development technology collection; these tools serve as the foundation for the web app. This interface provides the user (beekeeper) the ability to view and monitor the bee activity data of a beehive from their Raspberry Pi device, including the beehive's metrics such as inflow and outflow of bees over time, temperature, humidity, and wind-speed; the user has direct access to their beehive's data analytics and device status (online/offline).

The *MERN* stack was chosen because of its versatility and open-source nature, which matched the goals for this project. *ExpressJS* is the back-end framework, which helps to manage the routes and servers necessary to make calls to the back-end database, serviced by *MongoDB*. The back-end also utilizes *Mongoose* to model the data that exists in the database. On the front-end, the web application is primarly built using *ReactJS* and *JavaScript*, and utilizes powerful front-end development tools such as *Material UI*, a

React-component UI library, *Redux*, which manages the application's state, and *Axios*, which allows the front-end to make back-end requests. And lastly, *NodeJS* serves as this application's runtime environment for development and scalability.

The back-end and database structure can be broken up into three separate document models. The first document model is reserved for users. Users are comprised of a name, an email, and a password, which is hashed using *BCryptJS*, upon registration. When a user logs in, *JSON Web Token* is used for authorization across the web app. A user must be 'authorized' to access certain routes, such as the dashboard, as well as make certain requests to the API. An example of each field and datatype for the User schema is described in Table 3.1 along with its requirement and uniqueness.

Field	**Type**	**Required**	**Unique**	**Example Value**	**Description**
name	String	Yes	No	"John Doe"	The user's name.
email	String	Yes	Yes	"johndoe@example.com"	The user's email address. Must be unique.
password	String	Yes	No	"$2a$10$V7X9I...jMx7"	The user's password.
createdAt	Date	No	No	"2022-01-01T00:00:00.000Z"	The timestamp for when the user was created
updatedAt	Date	No	No	"2022-01-01T00:00:00.000Z"	The timestamp for when the user was last updated

Table 3.1: User Schema Table

The second document model is built for apiaries, each of which include a name

(unique), a location, a members array, and a devices array. The location is defined by a geographical sub-schema that includes spherical coordinates to determine a location for the apiary. The members array is another sub-schema that contains a reference to a user, and a specification if they are an owner of the apiary, or not (a viewer). Users specified as an "owner" of an apiary have the ability to edit that apiary; i.e. change its name, add/update/delete members, and add/update/delete devices. And lastly, each device in the devices array includes a serial number (from the Raspberry Pi device), a name, and time-series data provided by the machine-learning team. The data provided from the machine-learning team contains five main properties, a date provided from the system itself, raw x and y coordinates of the bees, temperature, humidity, and wind speed provided from a weather API, predicted x and y coordinates of the bees, and a last predicted deviation provided after multiple data points have been calculated. The first four properties are required with the deviation being left as optional because you cannot calculate this number without first receiving enough data points. An example of each field and datatype for the Apiary schema is described in Table 3.2 along with its requirement and uniqueness; some data fields such as the location, members and devices fields have sub-schema datatypes that are described in following tables.

Field	Type	Required	Unique	Example Value	Description
name	String	No	No	”My Apiary”	The name of the Apiary
location	Object of type `geoSchema`	No	No	See `geoSchema`	The coordinates of the Apiary
members	Array of Objects of type `memberSchema`	Yes	No	See `memberSchema`	An array of members associated with the Apiary
devices	Array of Objects of type `deviceSchema`	No	No	See `deviceSchema`	An array of devices associated with the Apiary
createdAt	Date	No	No	”2022-01-01T00:00:00.000Z”	The date when the Apiary was created
updatedAt	Date	No	No	”2022-01-01T00:00:00.000Z”	The date when the Apiary was last updated

Table 3.2: Apiary Schema Table

The location sub-schema is another data object model that holds a location type (defaulted to ”Point”), point coordinates, a formatted address, and a place ID (provided by the Google Places API). An example of each field and datatype for the Location sub-schema is described in Table 3.3 along with its requirement and uniqueness.

Field	Type	Required	Unique	Example Value	Description
type	String	No	No	”Point”	The type of the location data. Default value is ”Point”.
coordinates	Array of Number	No	No	$[-73.9857, 40.7484]$	An array of longitude and latitude coordinates in that order. Indexed as a 2dsphere to enable location-based queries.
formattedAddress	String	No	No	”123 Main St, New York, NY”	The formatted address of the location.
placeID	String	No	No	”ChIJd8BlQ2BZwokRAFUEcm$_q rcA$”	The place ID of the location.

Table 3.3: Location Schema Table

The Member sub-schema includes a reference to a User object to identify the member in this array, as well as a role that can have the value of ”user”, ”admin”, or ”creator”; different roles have varying permissions for the apiary the user belongs to. ”Users” can simply view the data, ”admins” can modify the apiary, and the ”creator” can perform any of the previous functions in addition to deleting the apiary as a whole. An example of each field and datatype for the Member sub-schema is described in Table 3.4 along with its requirement and uniqueness.

Field	Type	Required	Unique	Example Value	Description
user	ObjectId	Yes	No	”61579f9a53e7e8f2916d47b1”	References a user object associated with the member
role	String	No	No	”USER”	The role of the member, must be one of ”USER”, ”ADMIN”, or ”CREATOR”

Table 3.4: Member Schema Table

The Device sub-schema includes various information regarding the device in an apiary, such as the serial number of the Raspberry Pi device, a name for the device, a remote link (deprecated), and a reference to a Data object that holds the device's time-series data. An example of each field and datatype for the Device sub-schema is described in Table 3.5 along with its requirement and uniqueness.

Field	Type	Required	Unique	Example Value	Description
serial	String	Yes	Yes	”ABC123”	The serial number of the device
name	String	Yes	No	”Device 1”	The name of the device
remote	String	Yes	Yes	”https://remote.it/ABC123”	The remote.it URL of the device
data	ObjectId	No	No	”6154353e7eb3aa3b1886d051”	The ID of the data object associated with the device
createdAt	Date	No	No	”2022-01-01T00:00:00.000Z”	The date and time when the device was created
updatedAt	Date	No	No	”2022-01-02T00:00:00.000Z”	The date and time when the device was last updated

Table 3.5: Device Schema Table

And lastly, we have the third document model, split between the Data schema and Data-point sub-schema. The Data schema holds an apiary object ID and serial number string in order to reference an apiary model to ensure the data being pushed actually belongs to a real monitoring device. There is also an array of data-points, which stores the data-points that are sent to the Data-point sub-schema from the machine-learning team. The Data-point schema includes a time, activity data, weather (temperature,

humidity, and wind-speed), and a prediction deviation. Each data-point is processed on the device using the machine-learning model, which then creates a data object based on the Data-point schema and pushes the point to the relevant Data object's array in MongoDB.

Field	**Type**	**Required**	**Unique**	**Example Value**	**Description**
apiary	ObjectId	No	No	"6154353e7eb3aa3b1886d051"	Reference to the Apiary model
serial	String	Yes	Yes	"ABC123"	Unique identifier of the device sending the data
datapoints	Array of `dataPointSchema`	No	No	See `dataPointSchema`	Array containing the data points sent by the device

Table 3.6: Data Schema Table

Field	Type	Required	Unique	Example Value	Description
time	Date	No	No	Date.now	The time the data point was recorded
raw_activity.x	Number	Yes	No	15	The x coordinate of the raw activity data
raw_activity.y	Number	Yes	No	13	The y coordinate of the raw activity data
weather.temp	Number	Yes	No	25.4	The temperature in Celsius at the time of recording
weather.humidity	Number	Yes	No	50.2	The humidity in percent at the time of recording
weather.windspeed	Number	Yes	No	10.3	The wind speed in meters per second at the time of recording
prediction_activity.x	Number	Yes	No	2	The x coordinate of the predicted activity data
prediction_activity.y	Number	Yes	No	3	The y coordinate of the predicted activity data
last_prediction_deviation	Number	No	No	2	The deviation between the raw and predicted activity data at the time of recording

Table 3.7: Data-point Schema Table

Tables 3.6 and 3.7 describe the Data schema and Data-point schema, respectively

including the datatype of each variable, its requirements and uniqueness, and an example value.

3.2 Physical Hardware Platform

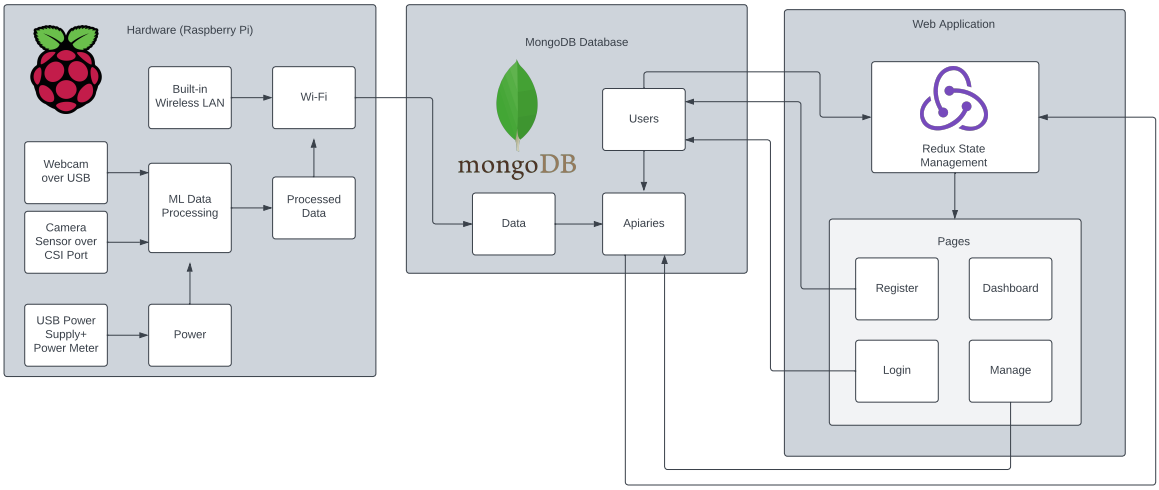

Fig. 3.1: System Information Flow Chart

The hardware we are using has all been selected due to its availability and open source nature. We have selected the Raspberry Pi as the main hardware platform, as they are well known to be easy to modify and use for custom systems such as ours. They are widely available for a reasonably low cost. The device that was settled on for the final design is the Raspberry Pi 4B. It is an extremely flexible platform, as it has built in USB, HDMI and CSI camera interface ports, alongside internal Wi-Fi and Bluetooth connectivity. It also has a large amount of processing headroom. The 4B has a 1.5GHz 64 bit quad-core CPU with 8GB of RAM.

Additionally, we are using inexpensive, widely available camera sensors that can interface directly with the Pi's CSI port, which is designed specifically for interfacing with imaging sensors. We are also designing the system to be capable of using a USB

webcam, as they are also inexpensive and widely available. We selected a few different camera modules to test, including the official Raspberry Pi HQ Camera, a 3rd party Smraza camera module, and a Logitech C615 USB webcam. The goal is to record in 1080P resolution at 30 frames per second, and all of these cameras support this standard. We settled on the Smraza camera for the final design due to its cost and flexibility for the setup.

The developed platform will communicate with the cloud processing servers using Wi-Fi, which will allow beekeepers to use their own Internet source. They can choose to use their own Wi-Fi or cellular hotspot to connect these systems to the Internet, giving them the flexibility to choose what works best for them. We used the built-in Wi-Fi module on the Pi, as it supports all modern Wi-Fi standards and allows for less extra cost compared to using an external adapter.

The system transmits the processed ML data over Wi-Fi to the MongoDB database, where it is stored in the appropriate apiary. Users can create apiaries to sort their devices, and allow multiple users to access these apiaries. It is common for an apiary to be managed by multiple beekeepers, so this function is essential. The web application displays this data, which is locked behind user accounts. Redux allows for state management, keeping track of where users are in the application and ensuring that data is only shown to users with the proper permissions. Although there is not much sensitive data being stored in the database, we still need to ensure that user's data is kept private and they choose who can access it outside of themselves.

Chapter 4

Evaluation

In this chapter we present the findings from the work we have done on the system through design, development, and implementation.

4.1 Web Application's Dashboard

In the original design for the web app, it was centered around a React-based application that required users to manage their own back-end, simply supplying them with the necessary means to manage their data. After design difficulties and some thought about the awkwardness of this solution, we decided to pivot the proposed solution towards a full-stack web application, built from back to front.

As a result of this change, the web application is built on the *MERN* stack, which utilizes *MongoDB* as the back-end database, *ExpressJS* as the back-end framework, *ReactJS* as a front-end library, and *NodeJS* as a JavaScript runtime environment for development purposes. The *MERN* stack was chosen because of its open-source nature, adaptability, and ease of development. It also allowed us to efficiently pivot from the original *ReactJS*-only infrastructure, to a full-fledged web-application. Users (beekeepers) are able to create an account, publish *Apiaries*, and setup devices that belong to beehives within an apiary. *Apiaries* can then be shared with other users, so multiple beekeepers can keep track of the same group of hives.

In the original mock-ups of the web app, there is a live video feed feature. To achieve

this, the Raspberry Pi's camera footage would be linked to the host of the Raspberry Pi. On an external computer, the Raspberry Pi's video stream can be watched by using the address and port of the Raspberry Pi. This can then be embedded directly within the dashboard through a JavaScript library. Persisting issues getting a stable video feed from the device to the application prevented us from fully integrating this feature, hence its presence in the original wire-frames, and absence in the final design.

In the figure below, the dashboard presents a video feed (top left), a controls section (bottom left), a quick-status section (top right), and a graph section (bottom right).

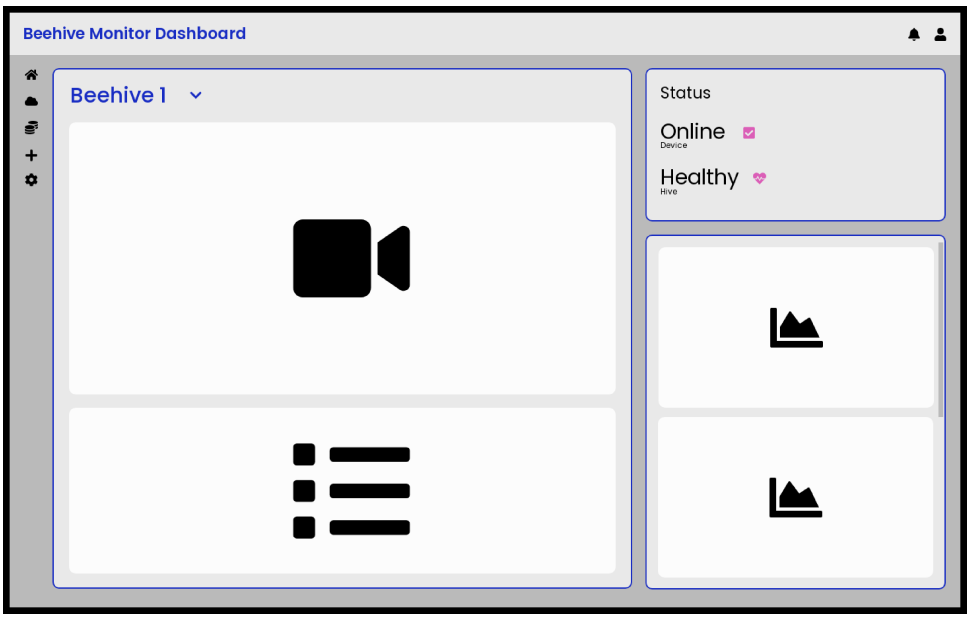

Fig. 4.1: Initial Dashboard Page Wire-frame Design.

The *Dashboard User Interface* also originally made use of Grafana's data visualization tools. The bee-count data would received from the machine-learning algorithm and saved in each user's own database with InfluxDB. To collect the data and input it into InfluxDB, the user would use Telegraf which can take files as inputs and save them into a database. The database stores the number of bees as an integer datatype, as well as the date/time as a timestamp datatype. Next, this database would be fed into a Grafana dashboard where it can be interpreted and displayed in a line-graph format. After the Grafana graph

has been set-up, it would be up to the user to directly embed it into the dashboard as a component of the app.

In the following figure, the user has the ability to adjust their user settings (left), add devices (middle), and embed their Grafana charts (right).

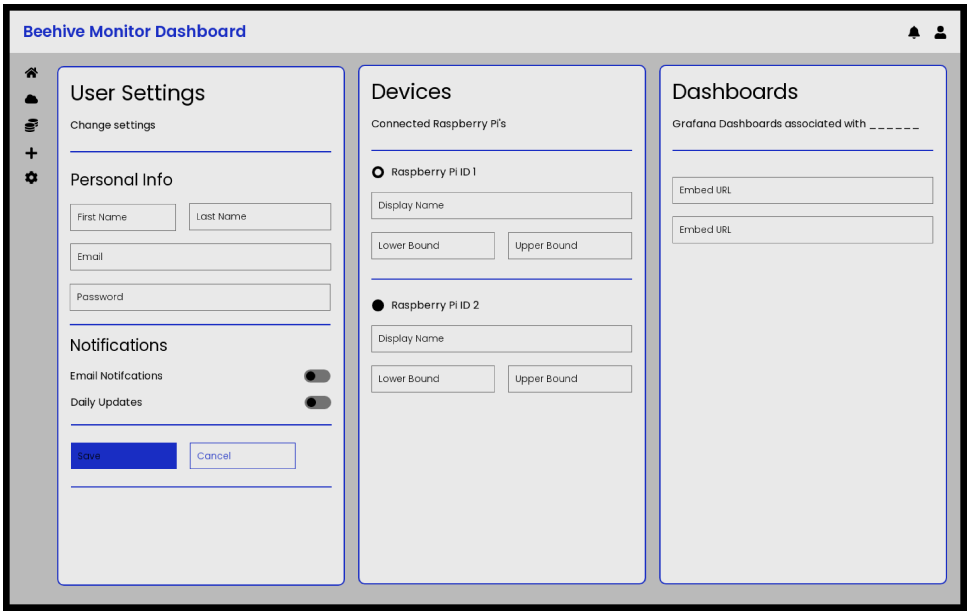

Fig. 4.2: Initial Settings Page Wire-frame Design.

The final design for the *Dashboard Web Application's* user interface was built using *JavaScript* and the *ReactJS* library. To assist in building a modern, user-friendly experience, we also utilized *Material UI*, a React UI tool that offers a comprehensive component-library designed for developers to quickly create production-ready UI components.

Another important tool on the front-end is *Axios*, which is used to make requests to the back-end to gather data from the database. In addition to *Axios*, the web app also utilizes *Redux* to manage the application's current state; this allows the app to manage what data and information it holds onto based on the current user, current route, and recent requests. And lastly, we utilized the *Recharts* React library to build an interactive

data chart tool that clearly and effectively displays the user's data.

In the application's final design, the dashboard underwent some changes compared to the original wireframe. The first major change is the lack of the live video section, which as previously mentioned, had technical issues that we were unable to overcome before the completion date. With this, we also decided that it would make more sense to put more emphasis on the data analytics, which is what the user would be most interested in when viewing the dashboard.

In the figure below, the final dashboard design is shown; the data chart is displayed on the left, with the quick-status section on the right. The data graph can be filtered by date/time ranges, as well as by metric (bee activity, humidity, temperature, wind-speed). The graph can also be scrubbed through to find more precise information about each data-point on the graph. The quick overview section informs the user if their monitoring device is online, and how many bees have been in and out within the past 24 hours, which a good indicator of the hive's activity.

Fig. 4.3: Final Dashboard Page Design.

Another important page to the web application is the settings, or management page. This page was also altered, as it no longer needed some functionality from the original

wire-frames after pivoting to a full-stack application. Another major change that sparked this change is the addition of ”apiaries”, which allows the user to organize their monitoring devices according to their actual apiary of beehives. This also allows users to add other members to their apiaries so that multiple beekeepers can monitor the same set of beehive devices.

The two figures below demonstrate the management page. Each apiary is organized as a card that can be expanded to access a list of members and associated monitoring devices. Device and members can be added, updated, or deleted, depending on the user's privileges for the apiary (User, Admin, Creator).

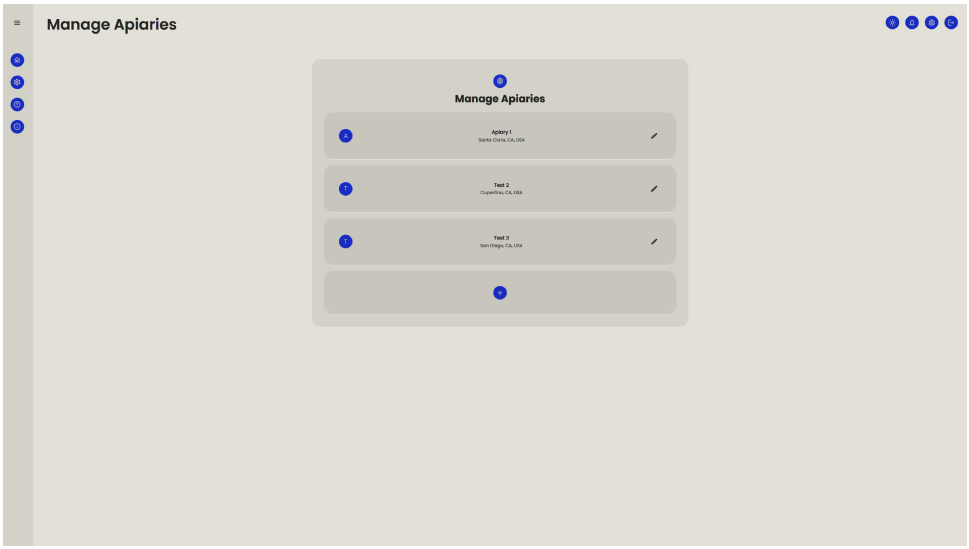

Fig. 4.4: Final Manage Page Design 1.

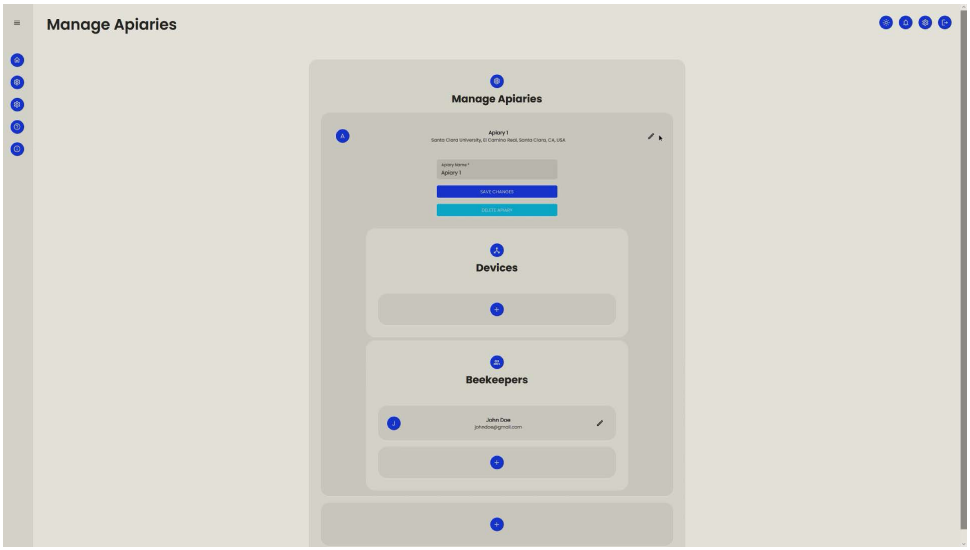

Fig. 4.5: Final Manage Page Design 2.

As mentioned before, the back-end is built using *MongoDB*, *ExpressJS*, and *NodeJS*. *ExpressJS*, which runs within a *NodeJS* server, is responsible for communicating with the back-end database using HTTP requests/responses, and bringing that information up to the front-end interface. The *MongoDB* database stores all of the necessary information for the users, including their registration credentials, apiaries, devices, and data points. This information is divided between three separate document models, and can be accessed through API calls.

As the back-end populates with data, we found that the front-end had trouble keeping up with the amount of data that was necessary to fill the dashboard's graph. So we made necessary alterations to the back-end functionality to limit the amount of data that is pulled when making GET requests to pull data into the interface.

Across the front-end and back-end, the change to create a full-stack application proved to be a transformative decision for the web application. By seamlessly integrating both ends of the application, we unlocked a multitude of benefits that enhanced the overall functionality and effectiveness of the proposed solution compared to the original design/early implementation. The full-stack application allows for the gathering and pro-

cessing of real-time data from the beehives with remarkable efficiency, delivering valuable insights to beekeepers on demand. After completing the implementation of the web application, we found that the adoption of a full-stack architecture proved to be instrumental to the success of this project, opening new doors of possibility and revolutionizing the way beekeepers interact with their hives.

4.2 Camera Streaming

One of the most crucial functions of a beehive monitoring system is the ability to remotely monitor the beehive from anywhere in the world. This remote viewing allows beekeepers to keep an eye on the beehive and ensure everything is running smoothly. Unfortunately, video streaming is very resource intensive, and given the relatively low power hardware on a Raspberry Pi, it is difficult to establish a reliable stream that can maintain an acceptable frame rate.

The first approach that was executed was attempting to stream using a service called *remote.it*. This original approach essentially created a local website on the Raspberry Pi machine that had a video frame in the HTML document. Then, using packages downloaded from *remote.it* specifically designed for Linux, the Raspberry Pi was able to broadcast that website along with a video stream to the open internet. However, due to the limitations of Raspberry Pi hardware, the stream was very unstable, had an extremely low resolution, used up too much CPU power, and often times didn't display a video feed altogether.

We then shifted our focus to streaming to YouTube, since YouTube has an open API which should have been be easy to implement into the web application. However, due to limitations of Linux software and how it processes video, the Raspberry Pi was unable to stream video through the command line. Following this discovery, we tried shifting focus

to a software solution - *OBS*. *OBS* makes it easy to set up a live video stream and have it transmitted to websites such as YouTube and Twitch. However, *OBS* does not have native support for Raspbian, the Linux software which the Raspberry Pi is using. As such, we had to use an unofficial build of the software that bypassed system requirements to run. This effort was short lived as although *OBS* was able to launch, initiating the stream prompted the software to immediately crash with no explanation other than "You are using OBS on an unsupported operating system"

Next, we pivoted to streaming using sockets programmed in Python. We were able to successfully stream video feed over a local IP address, and expected implementation of streaming to a public IP address to be relatively straightforward. The video feed itself is 480p, which is a welcomed improvement over the original approach. However, maintaining an acceptable frame rate is an area for improvement. We discovered that part of the reason the frame rate is so low (2-10fps) is due to the way in which the Raspberry Pi processes video. As of the time of publication, video processing isn't included naively in Raspberry Pi. As a result, we had to install *FFmpeg*, which converts pictures to video. Unfortunately, we were unable to have *FFmpeg* output a greater frame rate. We needed to find a solution that has better video processing libraries.

The final solution, while rather obscure, involved downgrading the software running on the Raspberry Pi to *Buster OS*. This operating system comes with more efficient built in video processing libraries that can be called in python with the "import picamera" command. Then, we can send the output of the camera to an IP address which can be loaded in any web browser on the same network. We were confident that this would be compatible with the web application that was developed. With this solution, we were able to achieve an acceptable frame rate of 30fps with a resolution of 480p.

While executing research and development, power consumption and resource utilization was of utmost importance to us, given the relatively weak hardware of the Raspberry

Pi. Below findings regarding power usage and CPU utilization with the 4 solutions that were implemented for video streaming can be seen.

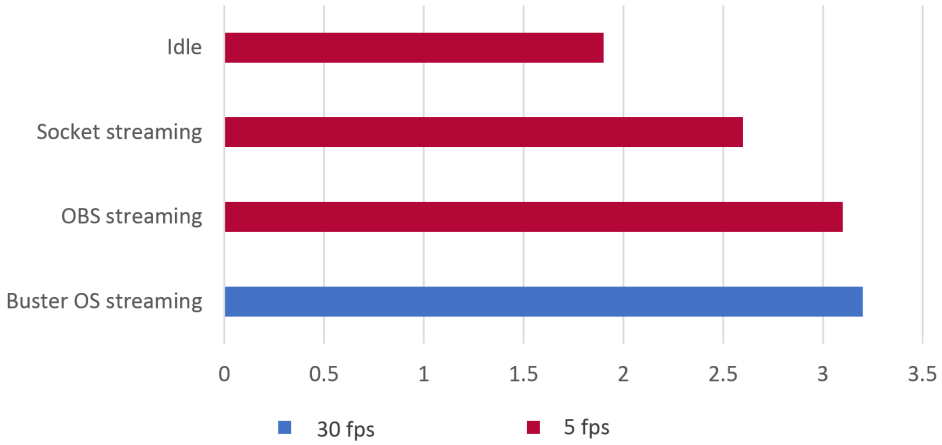

Fig. 4.6: Watt Usage

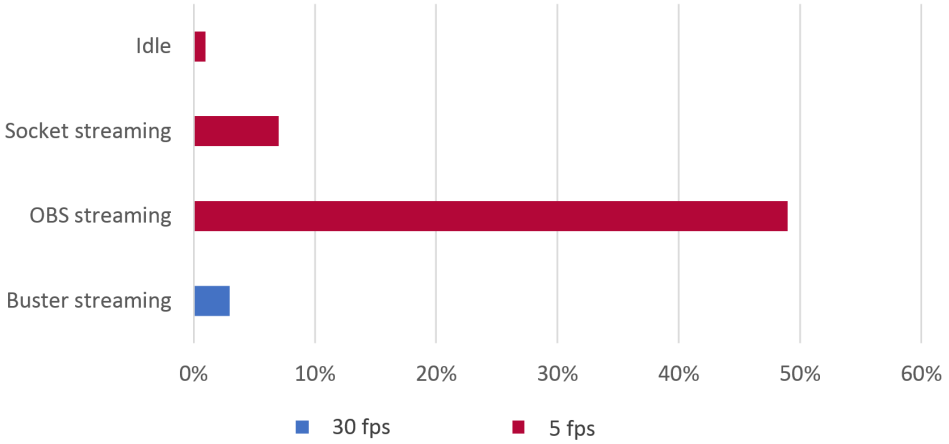

Fig. 4.7: CPU Utilization

In Figure 4.6 the solution with *Buster OS* only uses slightly more power than the other solutions, likely due to being a slightly older OS with less efficient power consumption. However, we found this power usage to be acceptable because we were able to maintain a 30fps stream. Figure 4.7 highlights how the final solution is CPU-friendly, only using about 3-5% at any given time which is only slightly more than the CPU utilization at idle. You can also see OBS streaming, even if functional, would not be an acceptable solution due to its extreme CPU usage.

Chapter 5

Future Work

In this chapter we present what tasks we want to have done for the system if given more time and resources to work on the project.

While this system is functional and achieves the goal of remote beehive monitoring, there are many improvements that could be made. One issue that was ran into during final deployment of the system was the fact that the Pi Camera library, which is used for both the live streaming code and the machine learning code, can only attach the camera to one process at a time. Due to the fact that the machine learning was developed separately from the streaming code, this issue was not discovered until the system was complete. This forced us to temporarily disable the live stream so that the machine learning code could use the camera, as this functionality is essential for the system to count bees. Future development could integrate the live streaming code into the machine learning code, so that they could run as a single process together and utilize a single Pi Camera virtual camera. While this integration would be ideal, another potential solution could be to attach two cameras to the Raspberry Pi, one for video streaming and another for video processing. Once this is achieved, the live video feed could be re-implemented into the monitoring dashboard, allowing users to easily view the entrance of the hive whenever needed.

Additionally, the remote Pi management is ran through Zero Tier, which is a subscription based service that is locked to a single user account. If custom Pi management code was created, this could allow remote management of the Pi through the web dashboard

without the need for an external service or subscription.

Another area of improvement that could be worked on is the hardware packaging. As we were unaware of how much processing power would be needed for the system, we utilized the most powerful Pi model available, the Pi 4B. However, after deployment, we discovered that the resource utilization was much smaller than anticipated. This could allow use of a smaller, cheaper, more efficient model such as the Pi Zero. If this is done, the size, cost and energy usage of the system could be decreased significantly.

In terms of adjustments to the software, there are many improvements to be made including various customizable features to user profiles, monitoring devices, and charts. When logging into a user profile if one were to forget their password there is currently no function to reset a password through an email verification or other type of personal verification. This is a key component to any user login interface because it is good practice to reset profile passwords every 90 days to constantly keep any account on the application secure. Within profiles, users should be able to add and remove themselves from apiaries but if a user is an administrator then they should have the ability to add and remove others from their apiaries. While monitoring devices, in the event of a relocation of the beehive the owner of the apiary should be able to change locations of the device. Currently this feature is not enabled but it should be included for prevention of this situation. There is also no solution to compare and contrast various monitoring devices at the same time. It might be important to know how two different colonies are performing at the same time and be able to view both of their graphs side by side. Regarding the graph, divisions of time on the x-axis of the graph can be proven to be important if you want to see a detailed view of the data. By including a more detailed view it allows the apiary owner to better understand intraday data points and compare that data to other days throughout a period of weeks, months, or years. The comparison between years is especially important because as seasons change year over year due to climate change it can be difficult to tell what

patterns bees should be following versus what they are actually doing. One last thing that should be included in the software is device analytics. Application owners should consider monitoring the Raspberry Pi's CPU utilization, memory allocation, storage capacity, packets sent and received over the network which would then be outputted to a graph on the web application. These analytics would help narrow down any issues that arise from troubleshooting the application so that a manager of an apiary can know that it is not the Raspberry Pi itself that is causing any issues. It would also be helpful to know when it would be time to upgrade to a newer version of a Raspberry Pi if a device slows down too much and the software is getting bottlenecked.

In addition to including new features, there should be consistent penetration testing of the application to discover any bugs that might come up and resolve them immediately in order to prevent disruption of service. Bugs can cause code to become insecure, program crashes, inaccurate data on the graph, data corruption or data loss, and integration issues with API's or hardware components. Therefore, in future implementations of this application, owners should develop significant tests to ensure every piece of the application works as expected so that when errors do arise they can be easy to resolve.

When deploying the application on Heroku, the service seems relatively inexpensive, stable, and easy to set up and use. However, future owners of the web application may choose to utilize Amazon Web Services (AWS) in order to benefit from Santa Clara University's (SCU) ongoing relationship with AWS and its ability to scale rapidly if there are too many users and too much stress on the application. This could be a possibility along with other hosting solutions if Heroku ends up becoming costly as the application scales.

As for the choice of using MongoDB for the database, the application is currently linked to a free cluster which might be inefficient later on if the application were to be scaled for enterprise use. Therefore, future application owners should look into purchasing a server subscription from MongoDB.

Chapter 6

Ethical Considerations

Arguably one of the most important parts of any new innovation is ensuring that it strictly abides to ethical standards. Given the project interacts with one of nature's most crucial linchpins, great effort was taken to design a system that would minimally disrupt the bee's natural behavior and habitat. This project has a responsibility to safeguard the well-being of a bee, and in large part the essential role they play in this ecosystem. By providing accurate information to the beekeeper, they can maintain healthy hives and avoid potential CCD.

By looking at this project from a high level overview, being able to have insight about a beehive's health, and more specifically how many bees are entering and leaving a beehive is very helpful to a beekeeper. This crucial information enables a beekeeper to more adequately care for their hive, which leads to healthier bees, and in the long run leads to a healthier ecosystem. Bees play a crucial part in making sure plants are pollinated; without them a lot of the ecosystem would die off. This alone is enough inspiration to build a solution.

Current solutions are quite expensive, often times costing upwards of $1000. This large upfront cost is something that needed to be kept in mind when designing this solution. By understanding that smaller backyard beekeepers cannot afford expensive beehive monitors it helped define the requirements needed for this project. In turn, this would hopefully cause a disruption int he market for other manufacturers to begin to develop open-source software that could be paired with low cost hardware to make

beehive monitoring more accessible.

As mentioned earlier, great care was taken in ensuring that the solution has little to zero interference with the hive, so that bees can go about their daily functions as usual. Some competitors have large cameras that obstruct the entrance of a beehive, which would disrupt a bees' daily routine. This solution has a tiny camera placed above the entrance of a beehive which in turn has minimal interference with the beehive and will not change how a bee behaves around a beehive. In terms of the privacy considerations of this camera, due to the downward facing orientation of the camera against a plain background at the entrance of the hive there is no possibility for any personal information to be viewed by the camera. When installing the camera it should be configured to avoid capturing any sensitive or private areas beyond the scope of the intended area to be monitored.

When it comes to storing data such as login information and data points in MongoDB, security and privacy considerations are important to take into account. It's imperative to ensure that the API keys for accessing the MongoDB cluster aren't leaked onto GitHub or shown anywhere publicly otherwise someone could potentially access the data or push incorrect data to the database. It is also important to secure the MongoDB account with a strong password so that nobody will be able to guess or get into the account. Even activating multi-factor authentication was a consideration when trying to secure the MongoDB account so that even if someone were to guess the password they wouldn't have access to the account. In terms of storing user login information, all of the passwords for each account were hashed and salted to make sure that they weren't stored in plaintext to be viewed easily in case passwords were to be reused in other applications. A future ethical consideration would be to include reminders every so often to remind users in the application to reset their account passwords to prevent potential security breaches.

Chapter 7

Conclusion

In this thesis, we proposed a creative and inventive solution to bring beehive monitoring to smaller beekeepers. By analyzing various beehive monitoring solutions, we were able to understand what is currently implemented and what is and isn't implemented already and for what price. Most of the pre-existing solutions are paid products and services that are too expensive for backyard beekeepers to afford. Those solutions also don't support the software being open source so that anybody can contribute to new updates regarding the monitoring application. The proposed solution would bring an inexpensive, easy to install, and simple to use monitoring device and application to the everyday beekeeper so that we can bring beehive monitoring to more people. Along with beekeepers, we are also keeping future developers in mind by creating an affordable and open source platform to host the beehive monitoring on so that future updates can be quick and abundant. Lastly, the machine learning model used in this monitoring device can provide novel technology to backyard beekeepers who often can't afford technology with innovative solutions.

Bibliography

[1] AG, "Why bees are so important to the environment." https://www.environment.sa.gov.au/goodliving/posts/2016/10/bees.

[2] H. Ritchie, "How much of the world's food production is dependent on pollinators?." https://ourworldindata.org/pollinator-dependence, Aug 2021.

[3] EPA, "Colony collapse disorder." https://www.epa.gov/pollinator-protection/colony-collapse-disorder.

[4] S. Sarah Myers, AgTech Marketing Manager, "5 ways to measure beehive health with analytics and hive-streaming data." https://www.sas.com/en_us/insights/articles/big-data/measure-beehive-health-with-analytics.html.

[5] T. Kelton, "Eyes on hives." https://www.eyesonhives.com/app/, May 2019.

[6] Arnia, "Remote beehive monitoring." https://www.arnia.co/.

[7] Broodminder, "Every hive counts." https://broodminder.com/.