¡¡¡¡¡¡¡ Updated upstream =======
¿¿¿¿¿¿¿ Stashed changes

Theta Beta Engineer Project Report

Foundation Color Identifier and Dispenser
University of California, Irvine

Akhil Nandhakumar, Allyson Lay, Dalen Avrin Smith,
Elizabeth Yancey, Emma Shin, Harmeet Singh, Ival Momoh,
Jay Kim, Richard Tokiyeda, Victoria Sun

November 17, 2025

Contents

List of Figures

List of Tables

Chapter 1

Abstract

The Foundation Identifier and Dispenser aims to develop a machine that is able to extract samples from a picture of a human to determine the shade of their skin, allowing the machine to dispense a corresponding foundation shade. The results produced should be both accurate and reproducable. Equipped with computer vision libraries and color correction algorithms, the system allows the user to take an picture alongside a reference color sheet, which has colors of known values. These captured values are processed to correct both camera bias, and lighting correction so that results may remain consistent regardless of lighting conditions during image capture. The system converts RGB values to LAB values, which are higher in accuracy in physical color mixing, as opposed to RGB, which is used to describe pixel colors. The program calculates how much of each color is needed to recreate the user's skin pigment. These pigments are then dispensed via a mechanical system comprised of a Raspberry Pi, servo motors, and syringes. This project demonstrates an application of computer vision in the cosmetic market to alleviate the burden of overconsumption and promote inclusivity.

Chapter 2

Introduction

2.1 Research

2.1.1 Background of Problem

Testing foundation colors can be a frustrating experience for many, who are unable to find the perfect balance. At the end of the day, no line of foundation can realistically provide colors that cater to every possible skin tone. The seemingly unresolvable desire for a perfect shade leads many makeup users to spend hundreds on shades that are "close enough." This leads to lots of waste, not just in money, but in bottles thrown out after purchase because the match was ultimately unsatisfying.

The beauty industry produces 120 billion packaging units per year and 95% of this these units are discarded, as opposed to recycled [**?**]. In addition, traditionally a custom skin matched foundation can cost anywhere from $60-100 per bottle, which leaves the consumers with the dilemma of whether they should take the gamble on the bottle that's just almost right versus breaking their wallet on a hand matched bottle. Our custom mixer machine offers the same accuracy in skin tone while also promoting cheaper makeup, sustainability, and waste-reduction.

Our foundation color picker abandons the concept of creating a set of discrete skin tones to choose from, instead, opting for custom mixed shades depending on the skin color detected by a picture. It also offers the unique ability to sample a shade without having to commit to a full sized bottle.

2.1.2 Existing Solutions

BoldHue provides an option for an AI powered foundation mixer. It matches skin tone through a smart wand that detects color. What they promised was millions of shades and the ability to store user's shades in profiles. However, their color detection methods provide room for lots of error because color isn't percieved the same depending on the lighting of the room, as well as the high cost of their machine.

Our product will have built in lighting correction that removes biases from the camera and uses the Macbeth Color Reference sheet as a set of control colors. The reference sheet is what will allow the machine to recallibrate it's understanding of what colors are being percieved.

2.2 Design Choices

2.2.1 Past Considerations and Scrapped Plans

Off-the-Shelf Syringe Pump System

An early design decision we explored was to use commercially available syringe pump system available for purchase from a third party. This idea was ultimately abandoned due to bulkiness of commercial pumps, as well as high cost. Although designing our own pump system required more work from our mechanical team to design the system from scratch, relying on an existing system would have limited our freedom, increased our costs, and taken away from the educational value of the project. Designing a custom syringe pump offered hands-on learning experiences in CAD design.

Quick-Swap Syringe Mounting Attachment

We also considered designing the Syringe housing to allow users to detatch and fill syringes in between uses. This feature was ultimately unnecessary as the syringes don't need to be manually pumped. As long as the moters can rotate the opposite direction, the syringes dont have to be removed to retract. The final design eliminated the syringe-swapping mechanism in favor of refilling via holding a paint vial under the syringe and letting the machine retract the pump to draw in fluid.

Single-Piece Printed Housing

Our initial housing was a single piece. However, most 3D printers that were available have a maximum build volume of 256 mm x 256 mm x 256 mm, and the housing dimensions we had for the initial design was too close to this limit. Printing the structure in a single piece also risks expensive failed prints, warping, and geometric constraints. We shifted to a housing unit comprised of three interlocking sections, with custom brackets to join the segments securely.

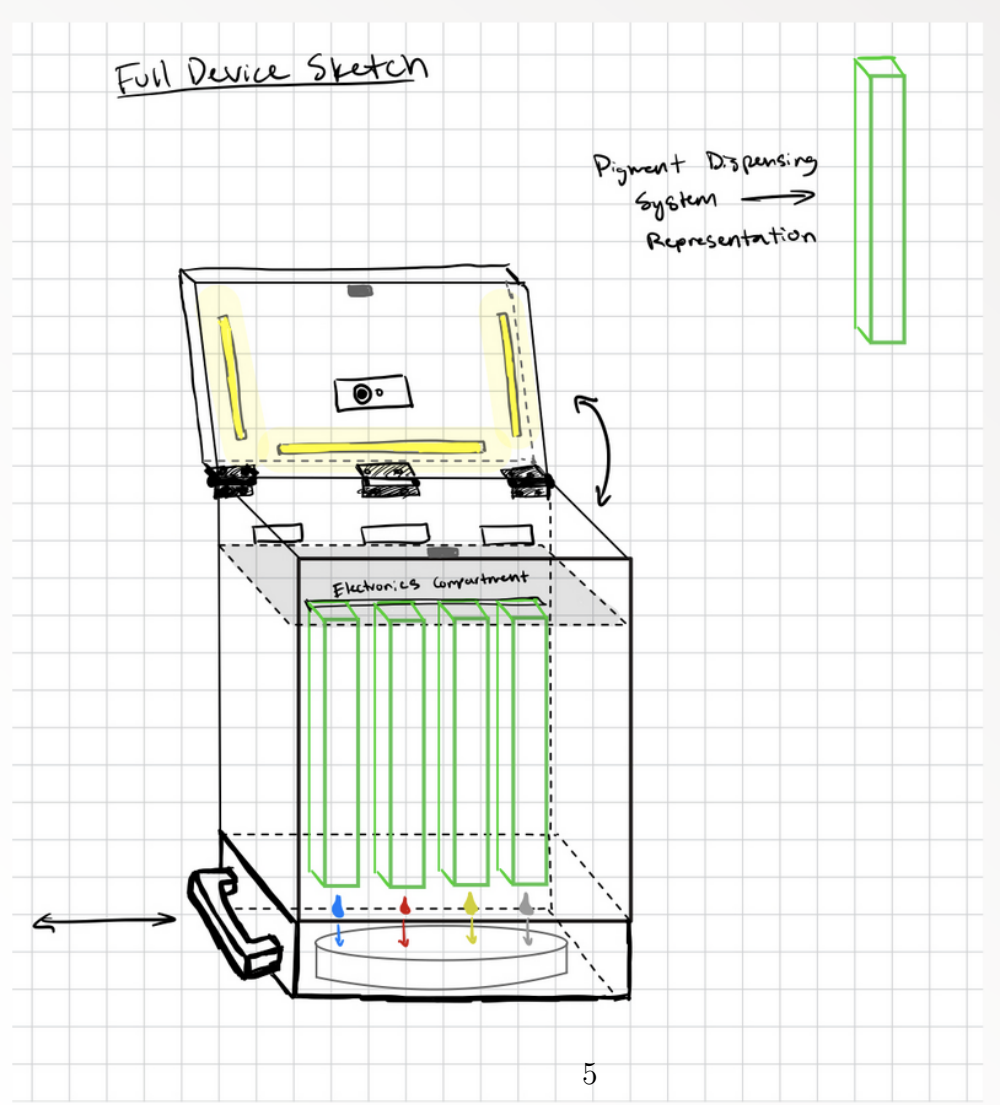
Chapter 3

Hardware Design and Specifications

3.1 First Iteration

3.1.1 Initial CAD Model and Hand Sketches

«««< Updated upstream



=======

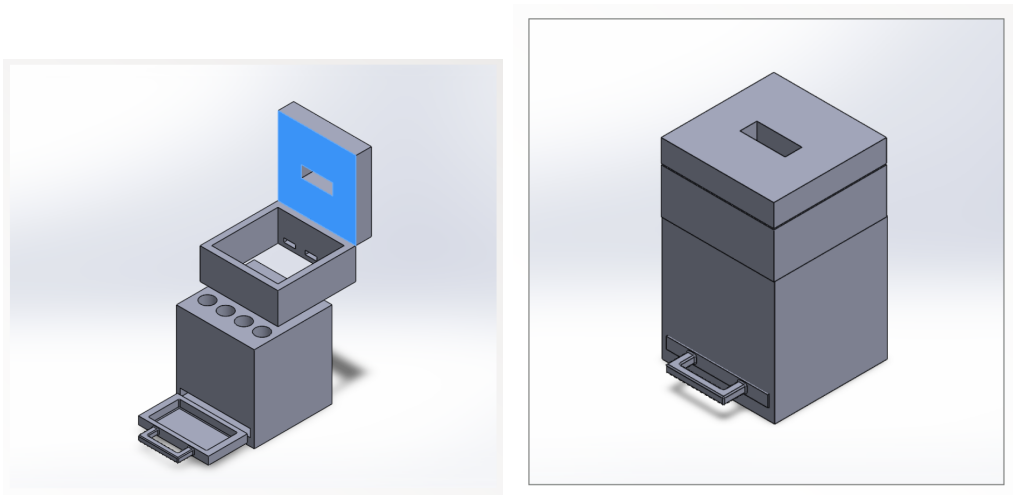


Figure 3.2: First CAD Models: Housing

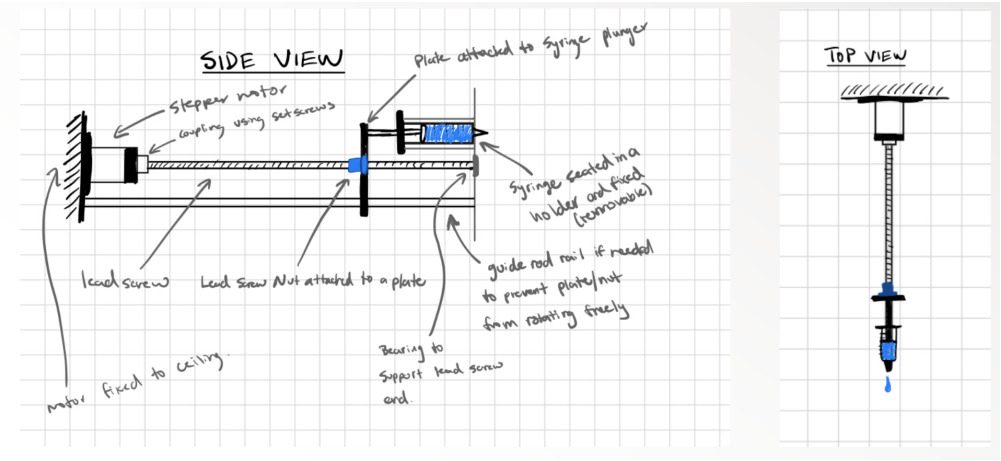


Figure 3.3: First Sketch: Dispenser Systems

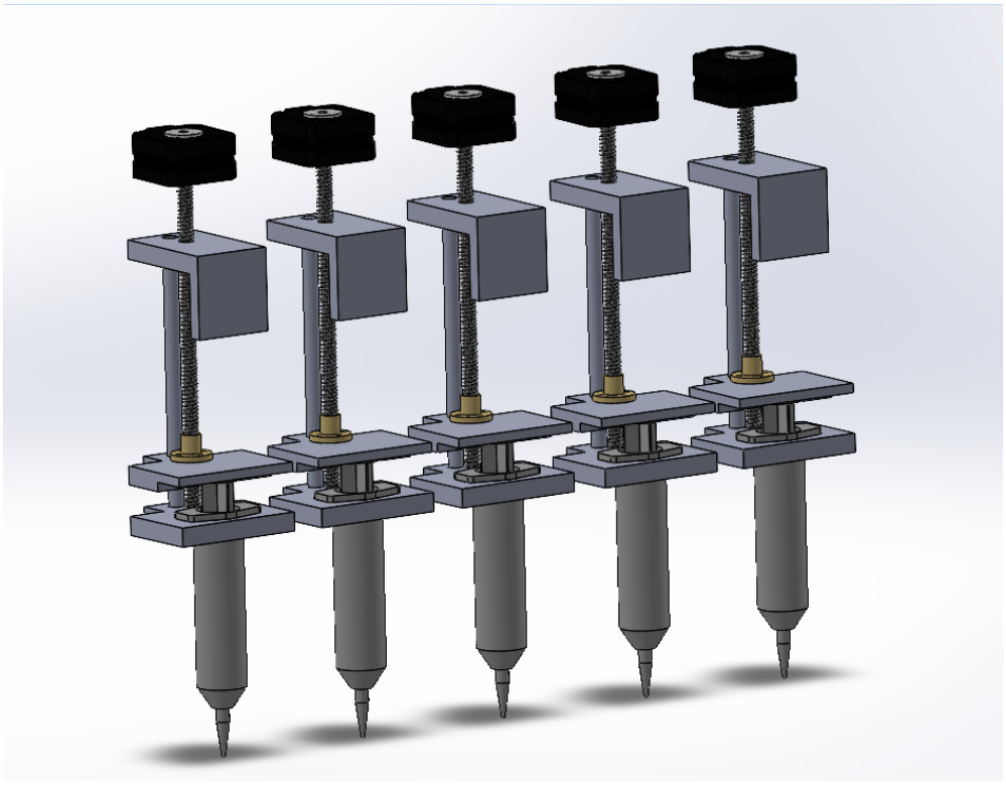


Figure 3.4: First CAD Models: Dispenser Systems

3.1.2 Commentary

3.2 Second Iteration

3.2.1 Updated CAD Model

3.2.2 Commentary

3.3 Third Iteration

«««< Updated upstream

3.3.1 Final CAD Model

3.3.2 Final Manufacturing Drawings for Custom Parts

3.3.3 Commentary

=======

3.3.4 Final CAD Models

Assembly

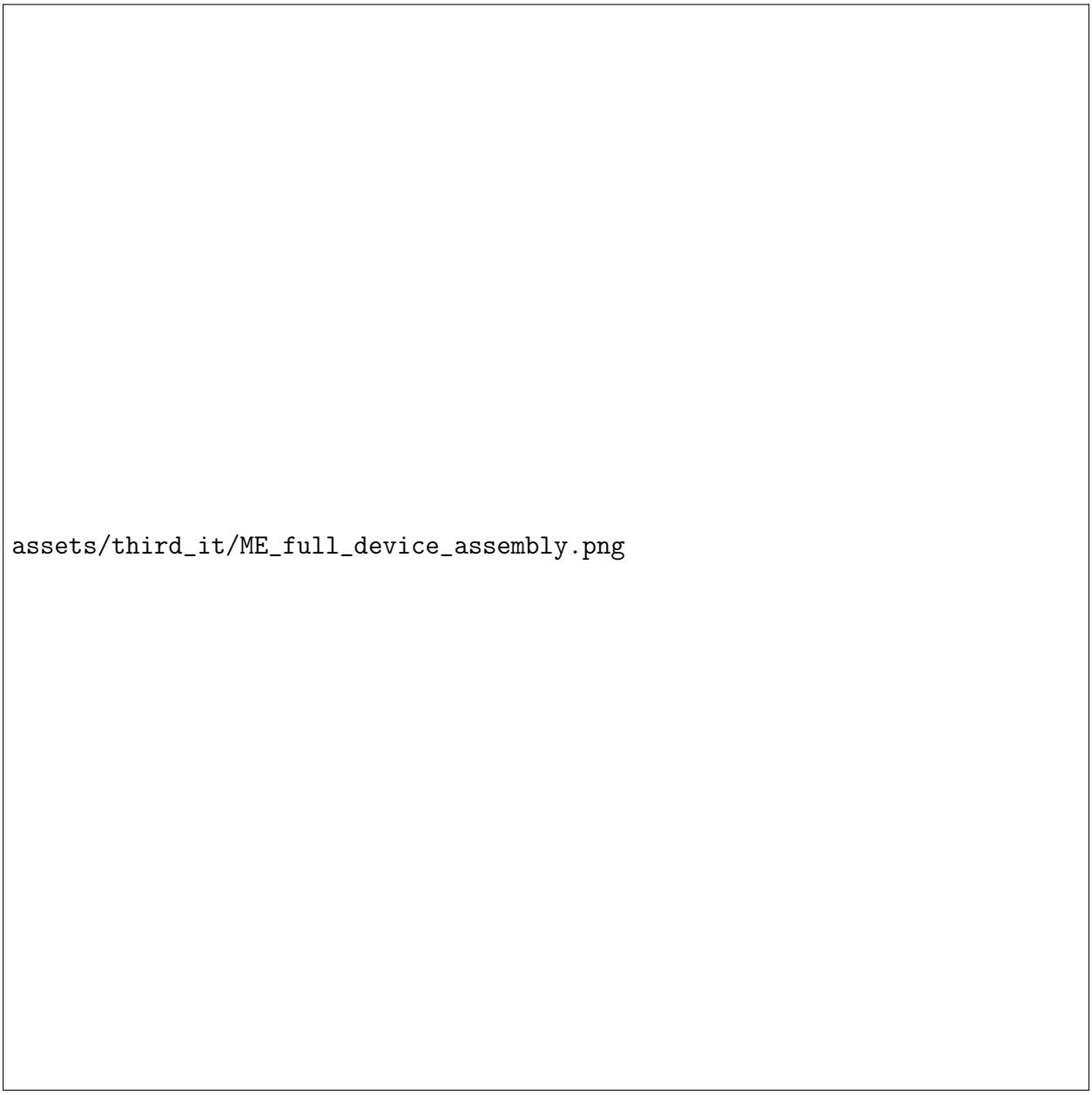


Figure 3.5: Final: Full Assembly *Exploded*

93 ***Housing***

Figure 3.6: Housing: Walls

Figure 3.7: Housing: Lid

Figure 3.8: Housing: Skeleton

The Housing walls were altered to include a slit because there are some screws in the skeleton of the product that would press up against the walls if we kept the second iteration's design. The slit was created to give the screws space and ensure that the components inside are not putting pressure on each other.

98 ***Dispenser***

Figure 3.9: Dispenser System : Brackets - for connecting to housing, guide shaft, stabilizing syringe, and pushing down syringe plunger

Figure 3.10: Dispenser System : Full assembly

The final design consists of four primary components: an L-bracket connecting the dispenser to the casing, a secondary L-bracket securing the guide rod, a syringe bracket that stabilizes and positions the syringe, and a plunger plate driven by the lead screw to depress the syringe plunger.

The final iteration incorporates several refinements across all components. Mounting brackets and L-brackets were redesigned with rounded edges to increase clearance between moving elements. The lower syringe bracket was modified to include two vertical extrusions that act as contact surfaces for limit switches. Additionally, mounting holes for fasteners were added throughout the assembly to improve structural stability and ease of integration.

3.3.5 Final Manufacturing Drawings for Custom Parts

assets/third_it/ME_d_exploded_drawing.jpeg

Figure 3.11: Full Assembly Exploded View

»»»> Stashed changes

Housing

assets/third_it/ME_d_Case_back.jpeg

Figure 3.12: Housing: Walls - Back

assets/third_it/ME_d_Case_Side.jpeg

Figure 3.13: Housing: Walls - Side

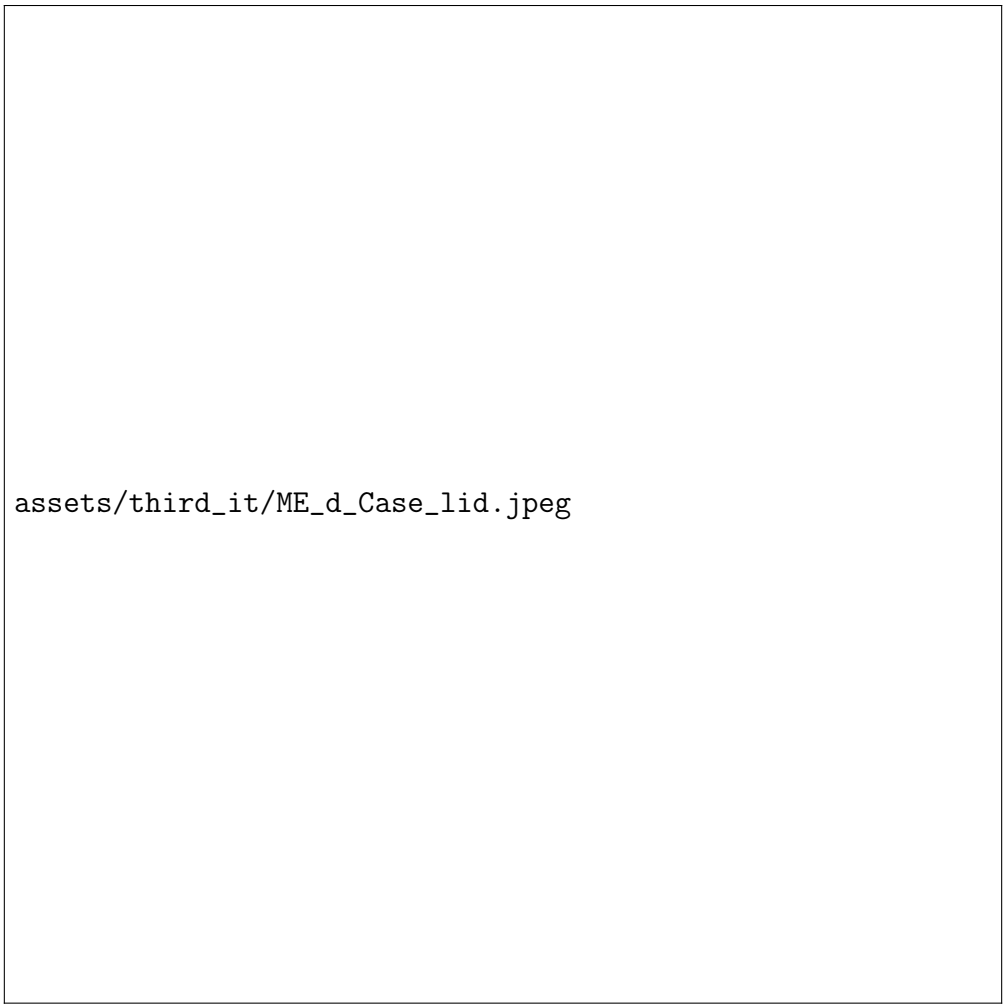


Figure 3.14: Housing: Lid

112 ***Dispenser***

assets/third_it/ME_d_Case_L_bracket.jpeg

assets/third_it/ME_d_L_bracket.png

assets/third_it/ME_d_Syringe_bracket.png

assets/third_it/ME_d_plunger_bracket.png

Figure 3.17: Dispenser System : Full assembly Exploded View

Chapter 4

Software Design

4.1 First Iteration

4.1.1 Preliminary Diagrams and Psuedocode

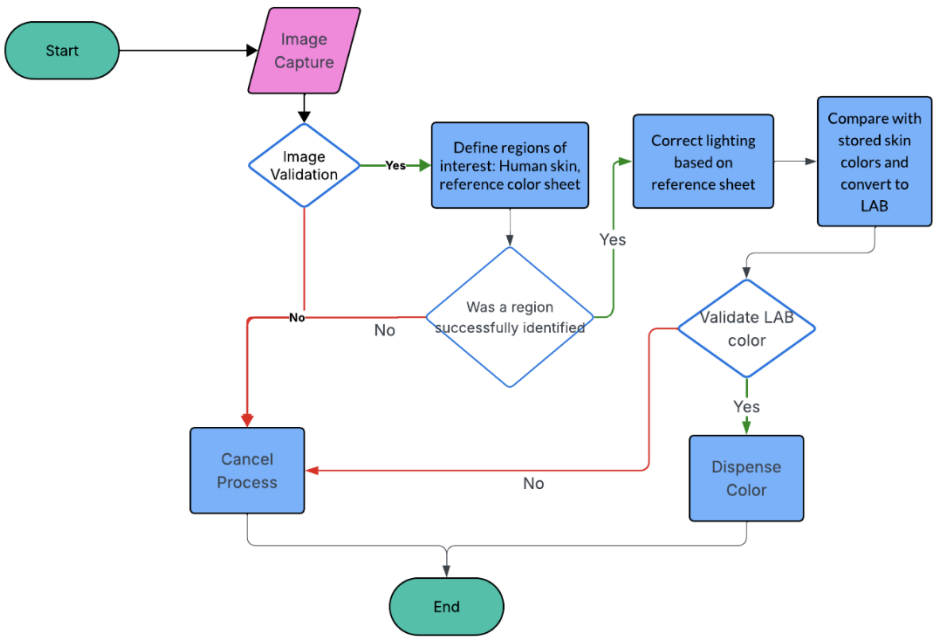


Figure 4.1: Initial Flowchart for Control Logic

«««< Updated upstream

```
# 1: Color Bias Adjustment and Sampling
```python
def IMAGE_PROCESSING(SAMPLE_PICTURE) :
    #INPUT: SAMPLE_PICTURE - image captured by camera
    #OUTPUT: LAB_values - color values compatible with paint mixing

    ### Get/Validate image containing skin region
    ### & reference colors/control sheet

    if SAMPLE_PICTURE is EMPTY/NOT_DETECTED :
        RAISE_ERROR "Image captured failed or canceled"
        return NULL

    # initial samples contain gamma-corrected RGB values
    # use OpenCV region of interest rectangle

    SKIN_SAMPLE = list of pixel RGB values that constitute a skin region from SAMPLE_PICTURE
    CONTROL_SAMPLE = list of pixel RGB values that encompass the reference color sheet SAMPLE_PICTURE

    # check whether the reference sheet is present or the image is too dark/light

    if CONTROL_SAMPLE is EMPTY/NOT_DETECTED :
        RAISE_ERROR "Contol sheet not detected. Take picture with color reference sheet in a well-lit room."
        return NULL

    # get average gamma-corrected RGB codes for the skin region and control

    SKIN_RGB_AVG = calculate average of SKIN_SAMPLE
    CONTROL_MEASURED_VALUES = list of averages of CONTROL_SAMPLE

    # get linear RGB codes from gamma-corrected RGB codes
    # allows linear operations to be performed on the codes

    SKIN_LINEAR_RGB = apply reverse gamma-correction to SKIN_RGB_AVG
    CONTROL_LINEAR_RGB = list of linearized RGB codes from CONTROL_MEASURED_VALUES
```

```
    # find difference in the control input RGB codes and stored RGB codes

    CONTROL_KNOWN_VALUES = load("CONTROL_DATABASE.csv") and linearize the codes
    CONTROL_TRANSFORM = find transformation matrix that makes CONTROL_LINEAR_RGB = CONTROL_KNOWN_VALUES * CONTROL_TRANSFORM

    # apply difference to the values found in the skin region for lighting correction

    XYZ = apply CONTROL_TRANFORM to SKIN_LINEAR_RGB

    # convert rgb value to lab for later processing

    LAB_VALUES = convert XYZ color space to LAB(XYZ)

    for element in LAB_VALUES :
        if element is OUT_OF_RANGE :
            RAISE_ERROR "color conversion error. take a new picture."
            return NULL

    return LAB_VALUES
```

Figure 4.2: Pseudocode: Image Processing Algorithms.
```

```
# 2: Raspberry Pi Code

```python
def EVENT_HANDLER():
    #extract lab values
    SAMPLE_PICTURE = CAMERA_CAPUTRE initiated by BUTTON_PRESS
    LAB_VALUES = IMAGE_PROCESSING(SAMPLE_PICTURE)

    #calculate servo turns for desired recipe
    PAINT_QUANTITIES =  assign paint quantities indicating how many
                        times the servo should turn with TARGET_SHADE

    DEPLOY_TO_RASPBERRY_PI(PAINT_QUANTITIES)
    #servo motors move syringes
```

Figure 4.3: Pseudocode: Embedded System.

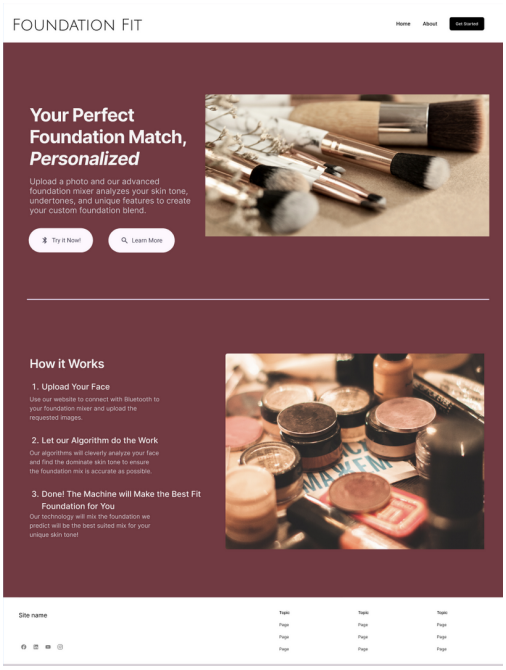


Figure 4.4: UI/UX - Landing page.
```

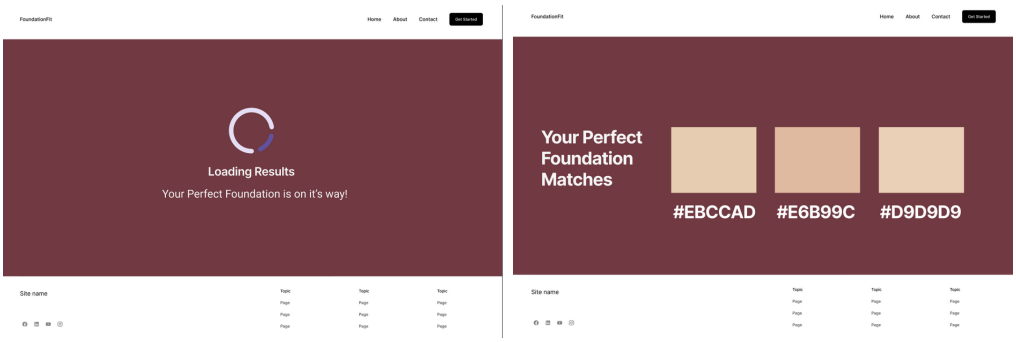


Figure 4.5: UI/UX - Loading the foundation shades.

4.1.2 User Flow Diagram

4.1.3 Techstack

Operating systems: Linux on Raspberry Pi
Libraries: OpenCV, NumPy, Pandas
Frameworks: React, Flask
Languages: Python

4.2 Second Iteration

4.2.1 Lo-fi/Mid-fi Designs

4.2.2 Basic Logic and Functionality

=======

assets/first_it/CS_pseudocode1.png

assets/first_it/CS_pseudocode2.png

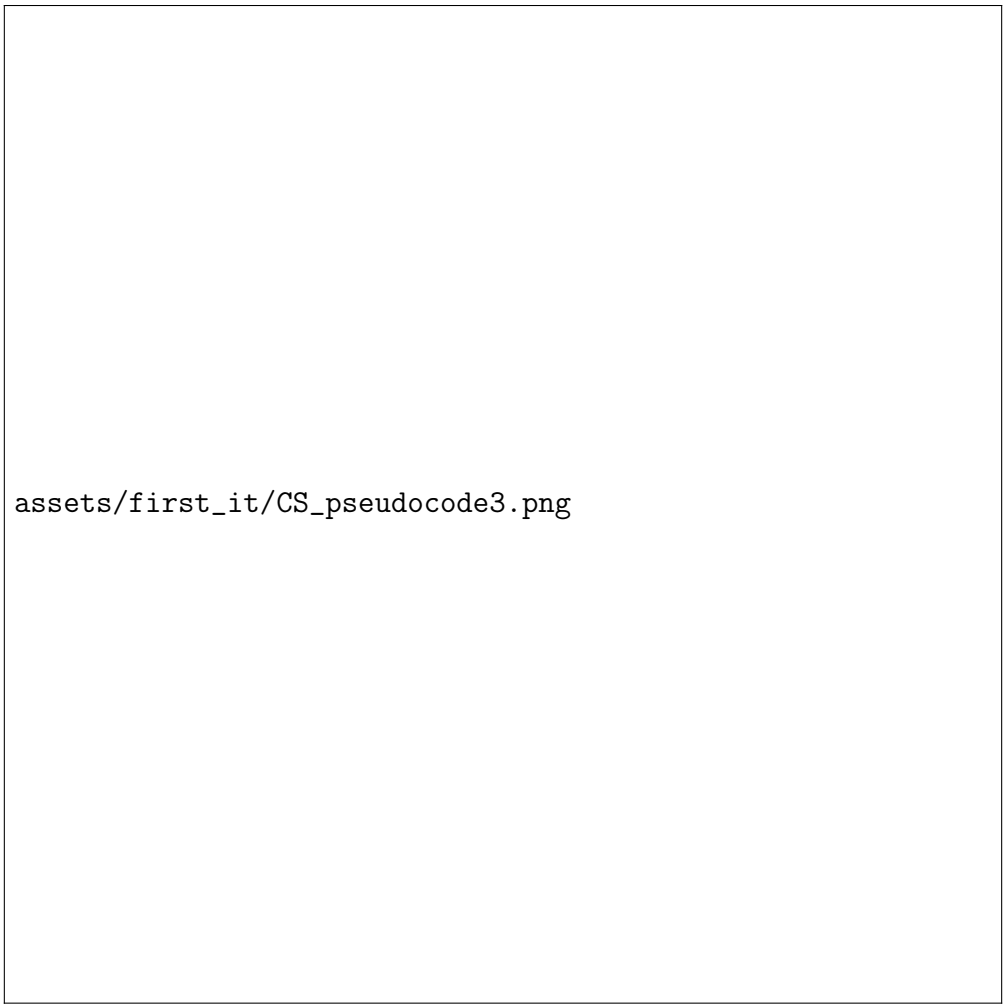


Figure 4.7: Pseudocode: Embedded System.

4.2.3 User Flow Diagram

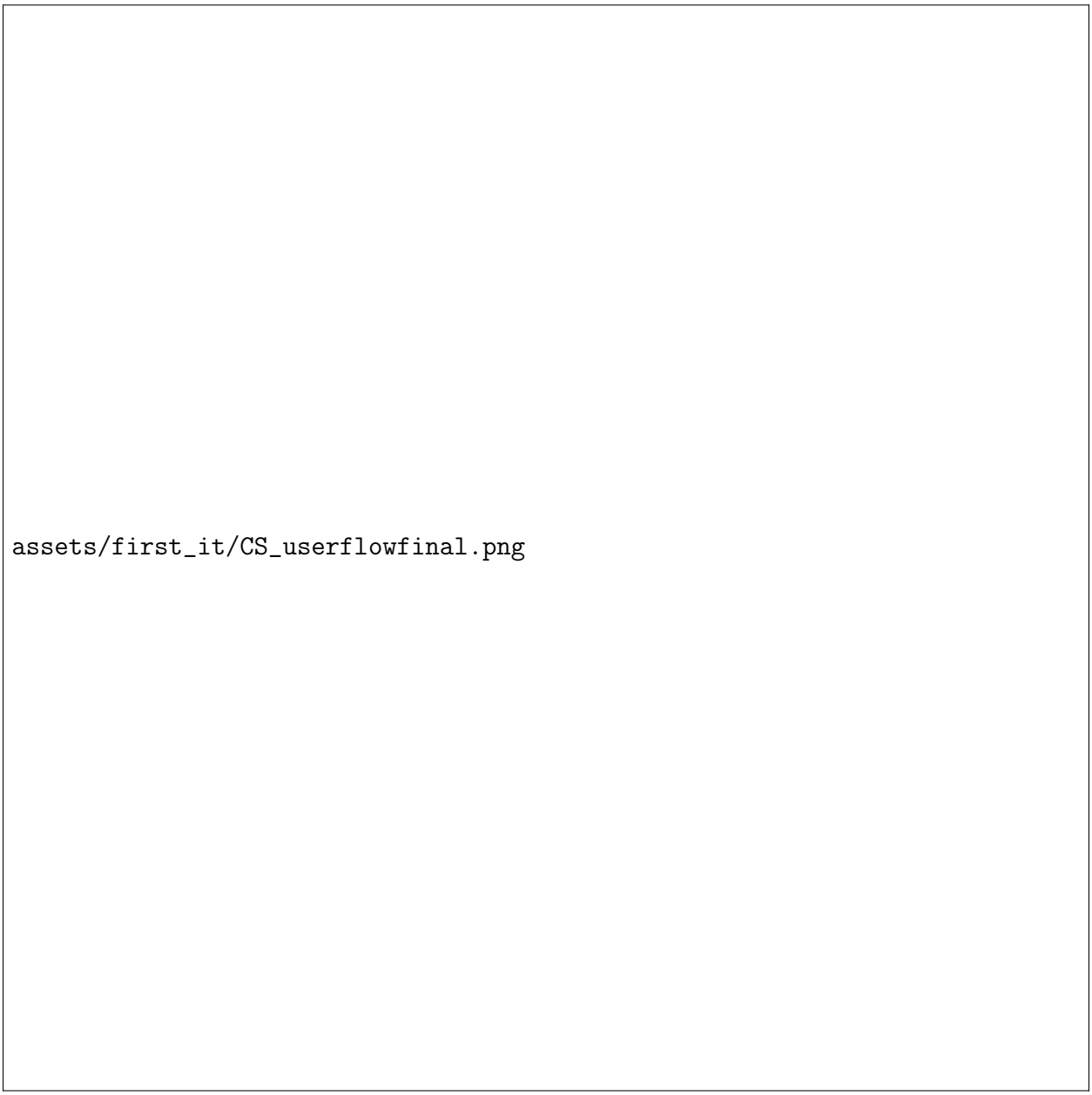


Figure 4.8: User Flow Diagram

4.2.4 Techstack

Frontend: React, Vite, Typescript, TailwindCSS, Lucide Icons
Storage : IndexedDB (local storage)
Hardware/OS/Hosting : Raspberry Pi (Linux, Pi Local Hosting)
Computer Vision & Data Science : OpenCV, NumPy, Pandas, Haar Cascades
Languages : Typescript, Python
Tooling : Git/Github

4.3 Second Iteration

4.3.1 Lo-fi/Mid-fi Designs

Figure 4.9: Initial Figma Mockup: UI/UX - Landing page.

Figure 4.10: Initial Figma Mockup: UI/UX - Loading the foundation shades.

4.3.2 Basic Logic and Functionality

The backend service is responsible for two core tasks: analyzing a user's skin tone from an input image and triggering the dispenser system for a requested foundation color. It is implemented as a Flask web API with three main endpoints: '/analyze', '/dispense', and '/ping' *app.py*.

Skin Tone Analysis Pipeline *'/analyze'*

Image intake (frontend → backend) :

Frontend allows the user to view a live feed and takes a picture when it detects the face reference sheet within the yellow box on the screen. The photo is sent to the '/analyze' endpoint. Backend strips, decods, and loads the data captured into an image object, which is saved for processing.

Skin region extraction :

The image is converted to a NumPy RGB array and passed to '`define_skin()`', which detects and isolates skin pixels. this focuses the analysis on relevant skin regions instead of the background.

Color space processing and lighting correction :

Skin samples are flattened into a matrix and normalized. '`gamma_to_linear()`' converts gamma corrected RGB values to linear RGB. '`lighting_correction()`' corrects the for inconsistent lighting and the corrected RGB is converted to XYZ in '`linear_to_xyz()`' then to LAB in '`xyz_to_lab()`'.

Output color encoding :

The average LAB color is mapped to an approximate RGB value in '`lab_avg_to_rgb()`' and formatted as a hexadecimal color string. This hexadecimal string is returned as '{"result":"<hex_color>"}' to the front end, which uses it for visualization for the user.

Dispensing Logic *'/dispense'*

Dispense accepts JSON payload containing the LAB code of the detected color. The endpoint validates that the color is detected and prints a message to let the user know dispensing has started. The endpoint is connected to the Raspberry Pi that drives the stepper motors and pumps.

HealthHealth Check *'/ping'*

The '/ping' endpoint is a lightweight health-check route that returns status updates. It allows the frontend to verify that the backend service is alive and responsive without invoking the full analysis pipeline.

»»»> Stashed changes

4.3.3 Core Features of Algorithm

The color detection algorithm transforms raw camera input into a corrected, device and lighting independent representation of the user's skin tone. This corrected color is then mapped to a cosmetic shade.

Macbeth Color Chart Reference Calibration

Figure 4.11: Macbeth Color Reference Sheet

The algorithm incorporates a Macbeth-style color chart for lighting and camera calibration. Known RGB or Lab reference values for each chart patch are compared to captured values:

- The color chart is detected and segmented from the image.
- We extract a sample from the middle of the patch.
- We correct the lighting and distortion

Robust Skin Sampling from Facial Regions

A Haar-cascade face detector identifies the face region and we extract stable sampling zones from the mask returned by the Haar-cascade. Regions of interest were defined as the forehead, nose, and cheeks. Multiple points inside each region are collected to avoid shadows, edges, or hair interference. These values are aggregated to produce a stable estimate of the user's skin tone.

Gamma Correction and Linear RGB Conversion

Since cameras produce gamma-encoded RGB values, the algorithm applies a gamma-to-linear transformation. Both chart colors and skin samples are linearized before calibration. Working in linear RGB ensures that subsequent lighting corrections and color transformations are physically meaningful.

Lighting Correction Using Chart Reference Data

A lighting correction is computed by comparing captured and reference chart patch colors. This correction is applied uniformly to the sampled skin pixels, producing values that approximate standardized, ideal lighting conditions.

Transformation into Lab Color Space

The corrected linear RGB values are converted into XYZ and then into Lab, a device-independent and perceptually uniform color space. Multiple Lab samples from different regions are averaged to form the final skin-tone vector.

Cosmetic Shade Selection and Hex Output

The final Lab vector is matched against a database of known cosmetic shades, each precomputed in Lab. The system selects the shade with the smallest perceptual difference (ΔE). The algorithm returns the shade as: a hex code, corrected RGB, and Lab code.

Integration with Hardware Dispensing Logic

The selected shade is mapped to pigment ratios for the dispenser hardware. The backend provides:

- `/analyze`: accepts an image and returns the computed shade,
- `/dispense`: controls Raspberry Pi motors to dispense pigment.
- `/ping`: returns status updates and process health

4.4 Third Iteration

4.4.1 Final Product

Foundation Fix Github

Chapter 5

Electrical Systems Design

5.1 First Iteration

5.1.1 Initial Wiring Diagrams

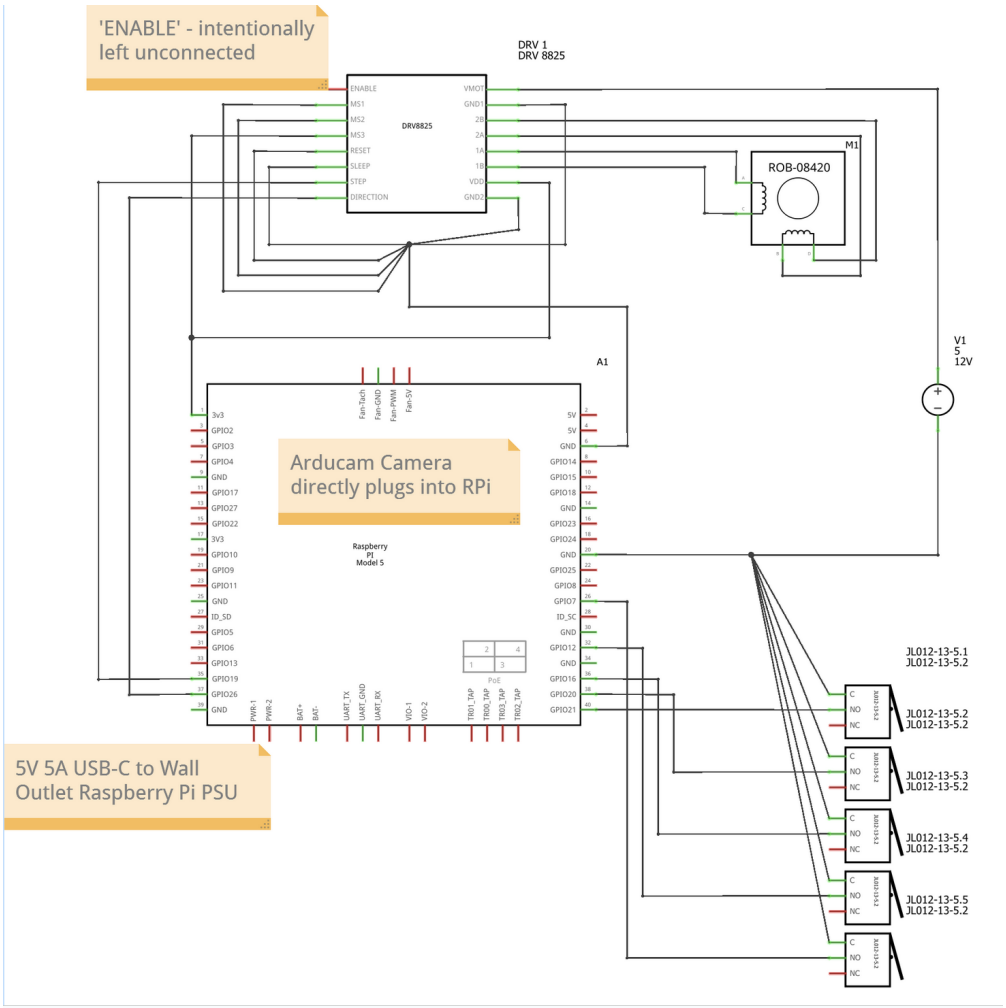


Figure 5.1: Preliminary Wiring Diagram

5.1.2 Initial Power Calculations

Parameter	Symbol	Value
Phase Current	I	0.7
Phase Resistance	R	4.0
Phases	–	2

Table 5.1: Power Calculations

$$P_{motor} = 2 * I^2 * R$$

$$P_{motor} = 2 * (0.7)^2 * 4.0 = 3.92 * 5 = 19.6W$$

$$P_{RaspberryPi} = 10W$$

$$P_{total} = 10 + 19.6 = 30W$$

5.2 Second Iteration

5.2.1 Circuit Building

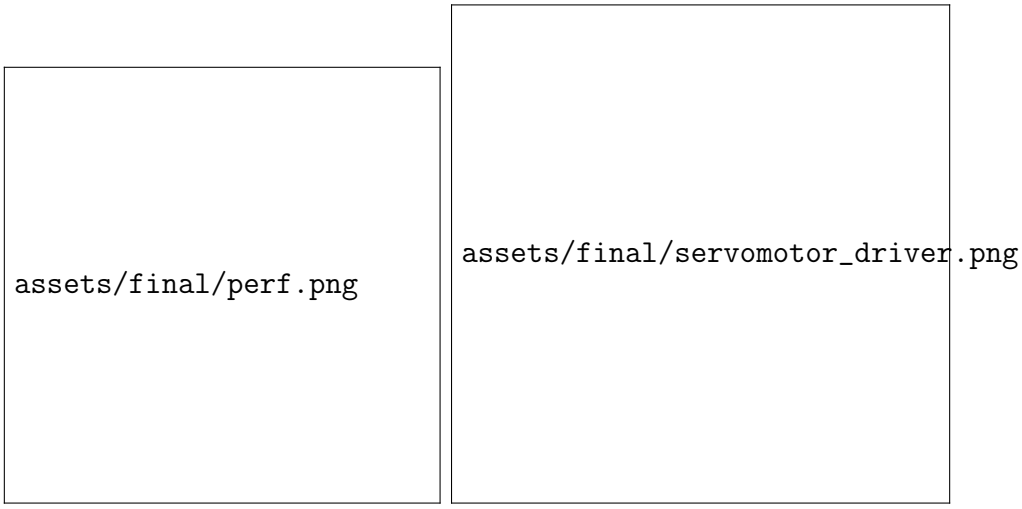


Figure 5.2: Perf Board and Servo Motor to Driver Connection

During the second iteration, several key changes were made to the electrical system. The original 12 V NEMA 17 stepper motors and their corresponding drivers were replaced with 5V 28BYJ-48 stepper motors paired with ULN2003 driver boards. This allowed the removal of the 12 V power supply and enabled the entire system to operate from a single 5V source. The wiring diagram was updated accordingly to reflect these component substitutions.

5.3 Third Iteration

5.3.1 Final Wiring Diagrams

«««< Updated upstream =======

assets/third_it/EE_second_wiringdiagrams.png

Figure 5.3: Final Wiring Diagrams

»»»> Stashed changes

5.3.2 Final Power Calculations

Per Motor:

Approximately 100 mA per coil 2 coils 5 V $\bar{1}$ W

Raspberry Pi 5:

Average consumption of 1.2A 5 V $\bar{6}$ W

Total System Power:

$$P_{motor} * 5 + P_{pi} = 1W * 5 = 11W$$

5.3.3 Circuitry

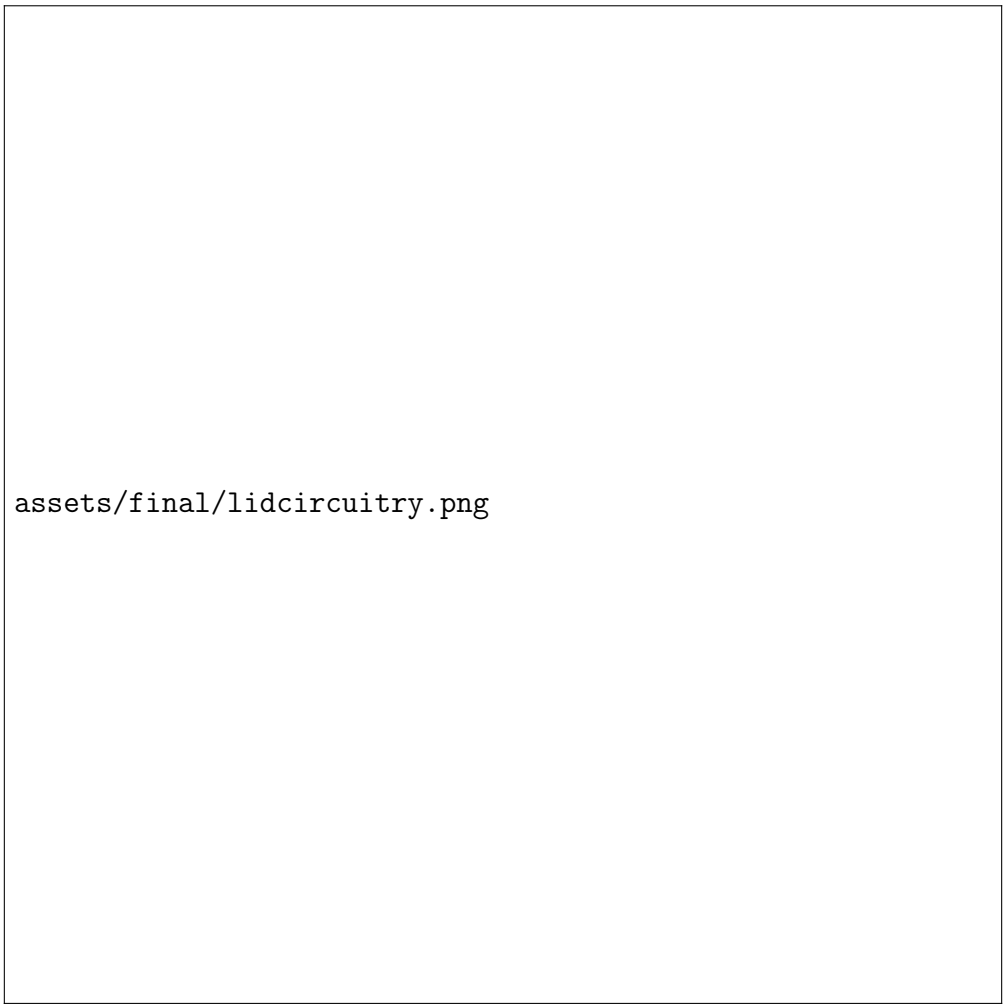


Figure 5.4: Final Circuitry Under Housing Lid

The only major change from the prototype stage was the transition from breadboard wiring to permanent soldered connections on a perfboard. This improved durability, reduced wiring instability, and prepared the circuitry for integration into the final enclosure.

Chapter 6

Final Product

Figure 6.1: Dispenser Systems Fully Assembled

Figure 6.2: Skeleton/Internal Structure

6.1 Testing Protocol

6.2 Integration

6.3 System Performance

Chapter 7

Discussion

7.1 Limitations

7.2 Future Work

Chapter 8

Conclusion

8.1 Bill of Materials

Component	Purpose	Qty	Price
2PCS 300mm Tr8x8 Lead Screw + Brass Nut	Linear motion	3	$13.99
5mm–8mm Lead Screw Coupler (5 pack)	Motor–screw coupling	1	$9.99
Linear Motion Rod Guide 8mm × 200mm	Linear guidance	3	$8.69
8mm Flange Pillow Block Bearing	Shaft support	2	$8.99
Raspberry Pi 5 (4GB RAM)	Main controller	1	$66.00
5 Pack Plastic 30mL Syringes	Fluid handling prototype	1	$6.99
Raspberry Pi 5 Power Supply	Power for controller	1	$15.99
Assorted Metric Fasteners (M2–M5)	Mechanical assembly	1	$20.00
Limit Switch (10 pack)	Motion limit detection	1	$5.99
Wiring Kit	Electrical connections	1	$6.98
5V 3A DC Power Supply	Low-voltage power	1	$7.47
5 Sets 28BYJ-48 + ULN2003 Driver	Secondary actuation system	1	$14.99
M2 Spacers	Mechanical spacing	1	$6.69
Perfboard / Breadboard	Circuit prototyping	1	$8.79
Mehron Liquid Makeup Colors	Testing material	1	$50.00
Large 3D Prints (>4 hrs)	Structural components	1	$144.00
Total Estimated Cost			**$441.12**

Table 8.1: Updated Bill of Materials.

References