

# ECSV311, 2024

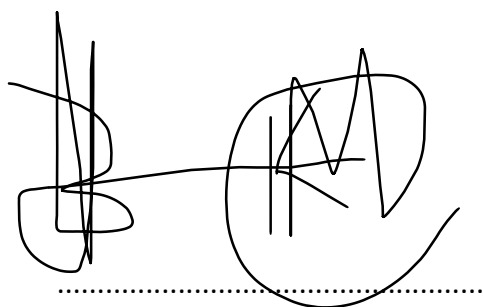
## Final Project

**Due Date:** 17 June 2024, 23:59

<b>Student Name</b>	K Mafurendi
<b>Student Number</b>	220702330
<b>Project Title</b>	NUCLEO Part Painting

### Declaration

I hereby declare that the work submitted by me is my own work and that I have observed the regulations prohibiting the copying of work from other persons and the observation of the rules regarding plagiarism.

A handwritten signature in black ink, consisting of a stylized 'K' followed by a large, circular flourish that encloses the letters 'M' and 'F'.

Signature

17 June 2024

## Table of Contents

1. Introduction .....	1
1.1. Abstract.....	1
1.2. Project Overview and Approach .....	1
2. Software.....	1
2.1. Control Algorithms.....	3
2.1.1. Buttonhandler – Pause/Continue .....	3
2.1.2. Estop Button .....	4
2.1.3. Production Mode state machine .....	5
2.1.4. The Pug painting and sorting algorithm.....	6
2.1.5. Hopper Tube handling .....	9
2.2. Error handling .....	10
2.2.1. Cylinder errors .....	10
2.2.2. Pug errors.....	10
2.2.3. Catastrophic errors .....	11
3. Protocol Implementation.....	12
3.1. NUCLEO to GUI.....	12
3.2. GUI to NUCLEO.....	13
4. Mobile device GUI and functionality .....	14
4.1. Home Tab.....	14
4.2. Stats Tab.....	16
4.3. Settings Tab.....	17
5. Operating Manual .....	18
5.1. Useful Tips.....	18
5.2. How To... .....	19

## 1. Introduction

## 2. Software

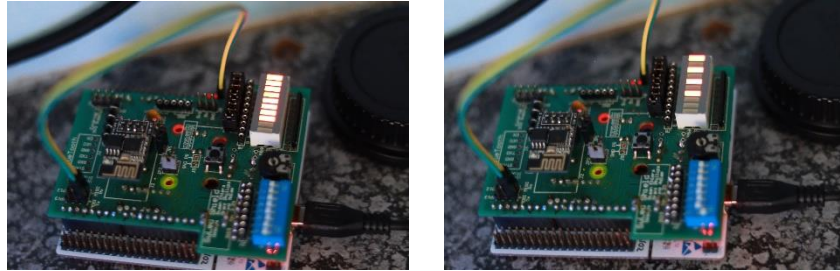
The bulk of the software is the Mbed code component to control the simulation and to handle MQTT communication with the cloud broker. The Mbed code can be divided into the following functional sections:

- **TCP and MQTT communication:** This section handles wireless communication via Wi-Fi using the ESP8266 and MQTT libraries. The section runs as a separate thread so that there is less interference or CPU conflicts with the simulation control code. Another reason and advantage of separating this functionality as a thread is so that the simulation control code can still run even when the NUCLEO is not connected to the internet.
- **Button input handling:** This section of code is part of the main thread that keeps listening for any of the used button inputs while the rest of the code is executing. The section is a custom function that handles both physical buttons and switches on the microcontroller or SWLed shield as well as mobile buttons on the GUI. The following buttons are handles in this section:
  - **Pause Button:** Physical this is SW0 on the shield and it has a mobile GUI alternative. When turned on (SW0 = 0 or mobile button pressed) the system pauses. When the system is paused, Leds 1 – 8 flash one after the other at a rate of 50ms continuously. These turn off when the continue switch has been switched on.
  - **Continue Button:** Physical this is SW1 on the shield and it has a mobile GUI alternative. When turned on (SW1 = 0 or mobile button pressed), if the system had been paused, it should resume back to where it was before being paused.
  - **Estop:** This is (PC13) the blue pushbutton on the microcontroller. When pressed, the system should half immediately. To resume, it would only have to be reset. When this button is pressed, Leds 1- 8 flash continuously at a rate of 50ms in a different pattern.
  - **Mode Button:** The system is designed to run in either of two modes, the test mode and the production mode.
    - **Test Mode:** A quick run through the entire process to test if all sections of the factory (parts of the simulation are working as intended). In this section mode, three pugs get loaded, painted and sorted one at a time. Each pug is painted a different colour to ensure that the paint gun can apply all the colours.

- **Production Mode:** This is the main mode, 8 pugs are loaded to the buffer wheel while being painted along the way, then they are sorted according to the desired recipe in the sorting hoppers

SW7 is used to switch between the modes where [OFF = Production mode, ON = Test mode].

When the microcontroller enters the production Leds 1 – 8 all turn on as shown below:



*Figure 1: NUCLEO in Production mode (left) and Test mode(right)*

Whereas in test mode, the every other Led from 1 – 8 is on while the other is off.

- **The main thread:** The main thread executes the simulation control code using a few finite state machines and a few supporting functions which are described in this chapter below.

## 2.1. Control Algorithms

### 2.1.1. Buttonhandler – Pause/Continue

Buttonhandler function flowcharts

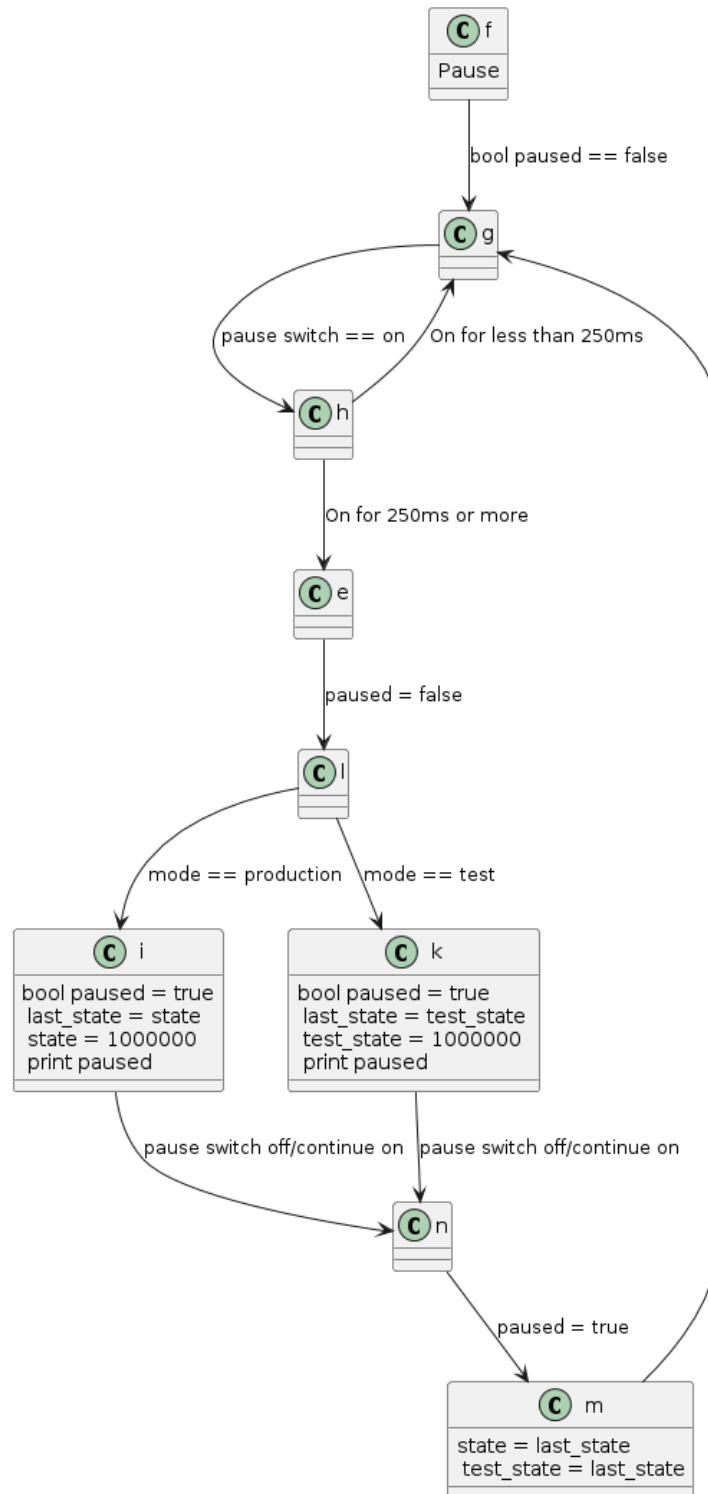


Figure 2: Flow chart for handling Pause/continue switches

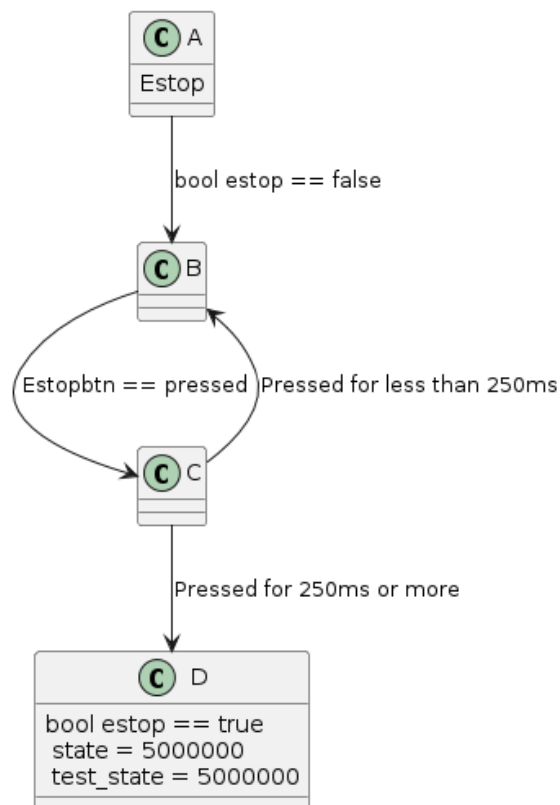
When the pause switch is turned on, if the system is not already paused, then the program goes into a pause mode. Whether the system is in a test mode or a production mode, the state machine will immediately transition to a 'pause state'. All the processes in the simulation are stopped and leds (Led 1 – Led 8) start to flash one at a time at a rate of 100ms for as long as the system is in pause mode.

Because a separate switch was used to pause then another one to continue (this was done to avoid conflict with the mobile buttons), the switch should be turned off before pressing the continue button.

When the continue switch is turned on, if the system was already paused, the system should continue right where it was before the pause button was pressed. The code snippet below shows how this was implemented in code:

### 2.1.2. Estop Button

**Buttonhandler function flowcharts**



*Figure 3:Flowchart for an emergency stop operation*

When the Estop button is pressed, the state machine immediately transition to an estop state. This overrides any operation or state that might be currently executing. The state machine stays in this state until the microcontroller is reset. The code implementation is shown below.

### 2.1.3. Production Mode state machine

The following 2 figures are a representation of the state machine that was used to control the simulation:

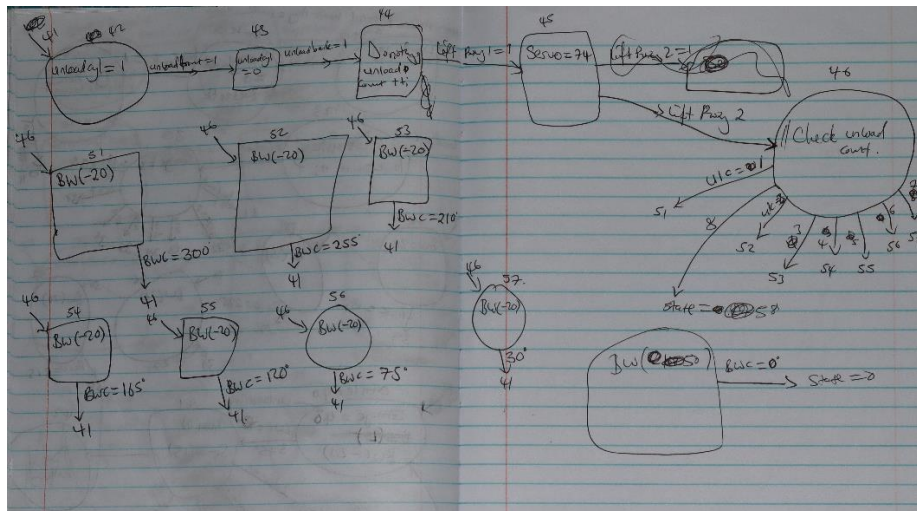
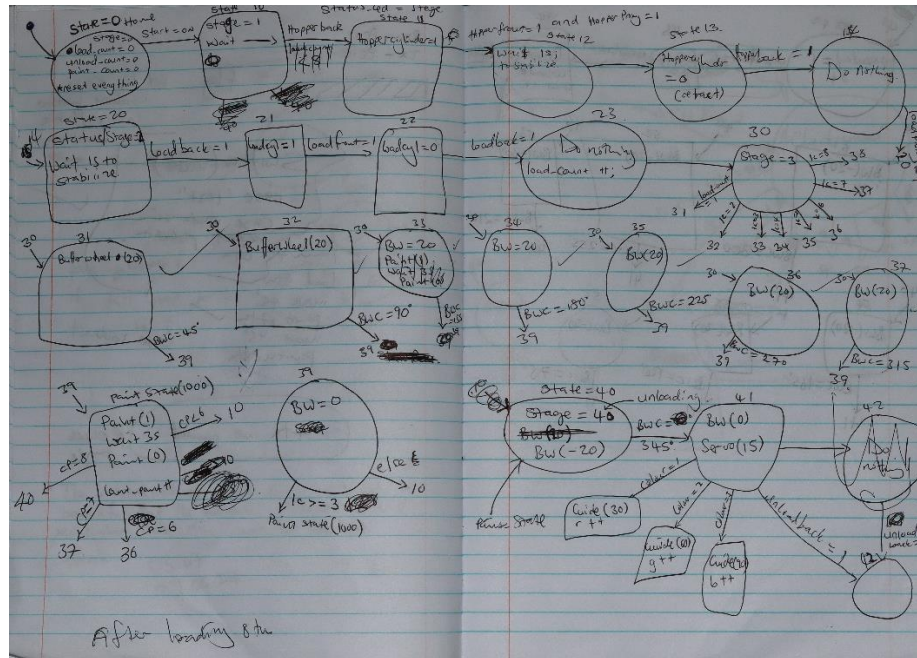


Figure 4: Finite state machine diagram for the Nucleo Part Painting Algorithm

Variables like buffer wheel speed, buffer wheel count, guide speed and count had to be figured by trial and error until reliable reading were found. This was the most challenging part but eventually, I found the following use of a do while loop and a range of angles where I would expect the buffer wheel counter value to fall in:

```
do
{
    np.BufferWheel(speed);
}while(np.BufferWheelCounter() < float(89.5) || np.BufferWheelCounter() > float(90));
state = 39;
```

*Figure 5: do while loop seeking the perfect angle*

State 39 is where the buffer wheel speed was set to 0.

A similar approach was also used for the guide counter:

```
float initialPos = np.GuideCounter();
ThisThread::sleep_for(4s); //allow pug to bre funnel into hopper and the initial position to be deduced
np.Guide(40);
while (np.GuideCounter() < initialPos + float(41.45)) { //40.5 work well, also .75 i guess
    ThisThread::sleep_for(10ms); // Small sleep interval for better control
}
np.Guide(0);
```

*Figure 6: While loop seeking the correct distance moved by the guide*

Storing the current position of the guide in the variable 'initialPos' ensured that I did not necessarily have to know the current position of the guide which made the calculation much easier.

Another area where great care was needed was finding balance between the speeds of either the buffer wheel or the guide motor and angles turned or distance moved respectively. Very high speeds not only made the pugs unstable and fall off the buffer but also introduced overshoots when trying to stop the wheel. The guide would also overshoot if the speed was very high and this would make the calculations more random and unreliable. On the other hand, a slow process would take too much time to perform a simple. I had to settle for a default speed of 10 for the buffer wheel and 50 for the guide motor. There is room however to adjust the buffer wheel speed from the GUI, although it was not implemented due to issues with MQTT library limitation which will be discussed in the GUI section.

#### 2.1.4. The Pug painting and sorting algorithm

Before the start button is pressed, the sorting recipe/sequence should be set from the GUI otherwise it will be 'rgb' by default. I used a function called init\_pug and a custom data type called pug. The pug data type was defined using the struct shown below which allowed easy allocation of color to each pug based on the recipe:



```
//Create a custom datatype called pug to represent each individual pug
typedef struct
{
    int pug_number; //basic number id for each pug from 0-8(for now)
    int colour; //colour representation of a pug [1,2,or 3] --Sort of, each pug decides its colour when it gets to the paint gun
    bool loaded; //state whether pug is loaded onto buffer(true or false)
    bool unloaded; //stet whether pug has been unloaded from buffer(true or false)
    bool painted; //state if pug has been painted(true or false)
    bool sorted; //state whether pug has been sorted into hopper(true or false)
}pug;
```

Figure 7: pug data type definition

In the init\_pug function, which is just a function to initialize the variables for each individual pug based on the provided recipe using a switch - case function. I ended up not using most of the variables due to time constraints but it could be used to improve the performance of the program. Nevertheless, this idea helped with a great deal abstraction. The pug is then declared as an array of size 30 (allowing a maximum of 10 pugs per sorting hopper tube maximum).

```
mode mode;
pug pugs[30];

case bi_gb:
{
    for(int i = 0; i < total_pugs ;i++)
    {
        pugs[i].pug_number = i+1;
        pugs[i].loaded = false;
        pugs[i].unloaded = false;
        pugs[i].painted = false;
        pugs[i].sorted = false;
    }
    for(int i = 1; i < total_pugs ;i+=2)
    {
        pugs[i].colour = 2;
    }

    for(int i = 0; i < total_pugs; i += 2)
    {
        pugs[i].colour = 3;
    }
} //end case
break;
```

Figure 8: Init\_pug function

In init\_pug, I literally assign a color to each of the 30 pugs using the iterations shown below. These are based on the manner in which my pugs were loaded and unloaded from the buffer wheel. The first pug to be loaded in a set of 8 pugs would be the last to be unload and the 2<sup>nd</sup> loaded was the first unload. This occurred because this was the easiest way for me to unload the pugs given the cumbersome angle calculations.

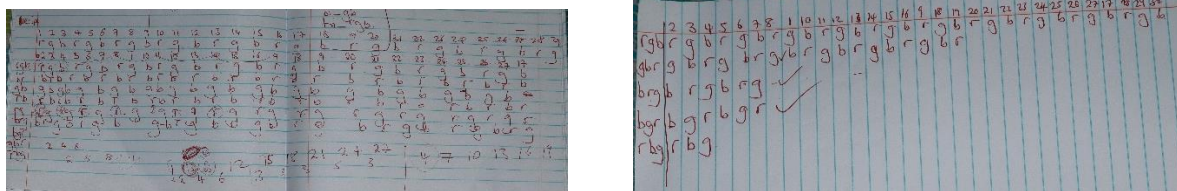


Figure 9: Calculations to assign color to individual pugs

This worked as expected and I was able to sort different recipes for different fill sizes:

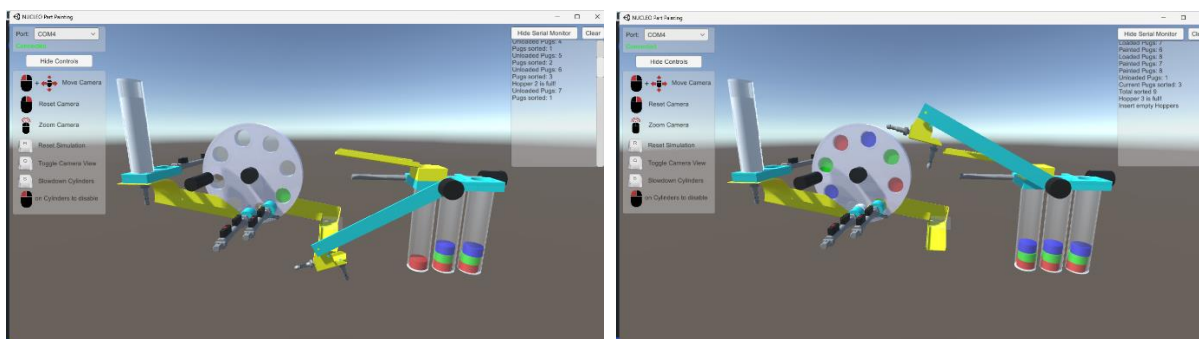


Figure 10: RGB recipe in action (fill size = 3 and Number of tubes = 3, color recipe = tri\_rgb)

If a cycle (of 8 pugs) ends and the sorting is not complete, another batch of 8 will be painted and pugs will be sorted until the sorting is complete as show in the 2 images above.

The flaw with this approach is that if a pug gets lost or fails to be painted the recipe is compromised because the system will unload the next pug in line whose color is not the one required for the recipe. It is also important to note that this flaw only affects multi color recipes. The precision of the guide can also compromise the recipe as demonstrated below.

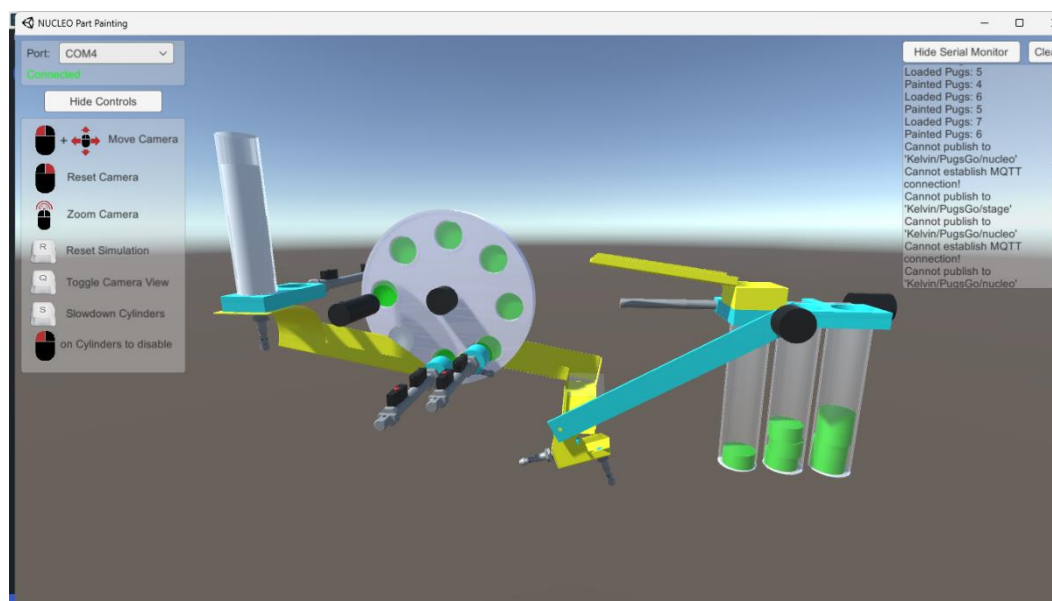


Figure 11: Demonstrating guide precision error (fill size = 4, Number of tubes = 3, color recipe = mono\_g)

This was supposed to be a `mono_green` (i.e., single color green) of fill size 4. Basically, this means the color sequence is G-G-G-G in each sorting hopper. Tube 1 was filled in as expected and the guide moved to tube 2 but when the last pug was unloaded it got stuck on the guide. This happens when the guide is not perfectly on top of the tube. Eventually tube 3 would end up with 5 pugs and tube 2 with 3 pugs.

In hindsight, I think that if sorting decisions are made when pugs reach the color sensor, it would have improved the operation significantly. My algorithm does not give room for these instances that compromise the sorting recipe.

#### 2.1.5. Hopper Tube handling

The same idea of a customer data type was used for the sorting hopper tubes to keep track of full hopper tubes as well as the movement of the guide motor.

```
//Define/Create a datatype called hopper_tube to represent a sorting hopper
//in the simulation with the following properties:
typedef struct
{
    bool empty;//true when the hopper tube is empty
    bool full;//true when the hopper tube is full
    int size;//the number of pugs required to fill the hopper tube
} hopper_tube;
```

Figure 12: `hopper_tube` data type

I also declared the `hopper_tube` as an array of size 3. I then used a similar function called `init_tube` to initialize the tubes.

```
void init_tube(int fill = 8)
{
    for(int i =0;i < 3;i++)
    {
        //tubes[i].tube_number = i+1;
        tubes[i].empty = true;
        tubes[i].full = false;
        if(fill <= 10 && fill >= 0)
        {
            tubes[i].size = fill;
        }
    }
}

//end func
```

Figure 13: `Init_tube` function

I could then use these and other variables to determine when and where the guide moves to in a separate function dedicated to handle hopper tube movements (`handletubes()`).

## 2.2. Error handling

### 2.2.1. Cylinder errors

Both in normal and slow mode, a cylinder in the simulation takes less than 5s to fully extend. So, if it takes longer than 5s there is most probably an error on the cylinder. In such cases, the user receives a notification that a particular cylinder has an error. If nothing is done, the system cannot move on.

If the cylinder is fix (say a cylinder had been deactivated and is now activated) the process will just move on from where it was.

This is the case with all the cylinders. It will be observed that each time a timer is started, I intentionally reset it first. This was to ensure that no other timing session is ongoing which might not have been reset, lest there is a conflict. An example of cylinder error handling on the load cylinder is shown in the code snippet below.

```
}
if(s.elapsed_time() > 5s)//faulty cylinder
{
    s.reset();
    sprintf(messages, "Load cylinder error!\n");
    np.printf(messages);
    if(np.LoadCylinderFrontSensor() == 1)
    {
        MSG = true;
        state = 43;
    }
    MSG = true;
    ThisThread::sleep_for(5ms);
    MSG = false; //so that MSG does not stay true and keep sending messages
}
```

Figure 14: Catching a cylinder error in code

### 2.2.2. Pug errors

A number of errors happen on the pugs given the unstable and random nature of the simulation.

- **Pug retrieval error:** A pug can fall off the platform upon retrieval or it can get stuck before reaching the loading area. If it does not make it in 30 seconds, based on trials, it is definitely stuck or lost. The system therefore will retrieve another pug. Depending on the nature of the error, this can solve the problem or it won't help. I discuss that in the catastrophic errors section below.
- **Lost Pugs:** Sometimes, a pug does get to the loading area but fails to get loaded and falls off and so on. There is going to be in that case an empty slot on the buffer wheel as it goes through painting and unloading. The pug will be recorded as lost at unloading. On the other hand, a pug might fail to get unloaded due to angle precision failure or otherwise, it will also be counted as lost. From unloading to the lift tray, if a pug based on trials, takes more than 45 seconds it is definitely lost. The system simply moves to unload the next pug in line.

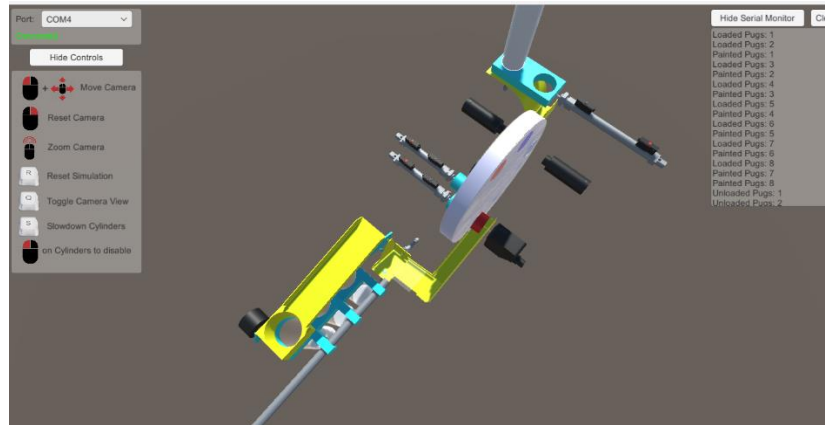


Figure 15: Pug unloading error

- **Sorting error:** A pug can successfully go through all stages but due to precision error on the guide motor, that pug may end up in the wrong sorting hopper or getting stuck on the guide funnel. Since there are no sensors to pick that, these errors will just go unchecked. I guess that is where the quality control team comes in.
- **Colour Sensor error:** On rare occasions, if a pug fails to unload and gets stuck in front of the colour sensors, the colour sensor behaves as if it is sensing a huge traffic of the same colour. This ends up distorting any colour counters or variables whose value depend on the colour sensor outputs.

### 2.2.3. Catastrophic errors

When a pug experiences a retrieval error and it get stuck in such a way that the next pug in line cannot push it off to the loading error or at least replace it, the only way out of this is to reset the system.

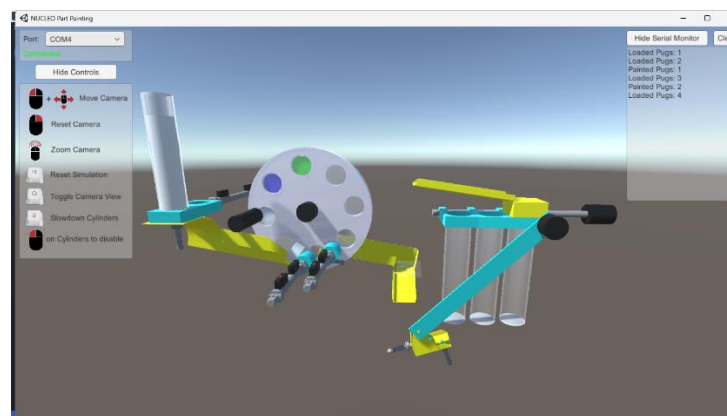


Figure 16: Catastrophic error

Pugs can also get stuck and accumulate after unloading causing another form of catastrophic error.

### 3. Protocol Implementation

An IoT MQTT Panel application was used for the graphical user interface, so communication is via a TCP/MQTT protocol. To establish communication between the GUI and the NUCLEO, the following MQTT topics were created. All topics have the home tab/ main prefix:

**Kelvin/PugsGo/...**

#### 3.1. NUCLEO to GUI

The NUCLEO publishes the following topics, then the GUI subscribes to these topics:

- **Home tab:**

The homepage gives a livestream of each cycle of 8 pugs:

<i>Topic</i>	<i>Publisher</i>	<i>Subscriber</i>	<i>Payload</i>
<i>loaded</i>	nucleo	GUI	An integer from 0 (loaded) pugs
<i>painted</i>	nucleo	GUI	An integer from 0 (painted) pugs
<i>sorted</i>	nucleo	GUI	An integer from 0 (sorted) pugs
<i>nucleo</i>	nucleo	GUI	1 tells GUI that NUCLEO is still live
<i>alert</i>	nucleo	GUI	1 or 0 turns on and off if Estop is ON
<i>msg</i>	nucleo	GUI	String literals. Specific message to GUI
<i>red</i>	nucleo	GUI	An integer from 0 – number of red pugs detected
<i>green</i>	nucleo	GUI	An integer from 0 – number of green pugs detected
<i>blue</i>	nucleo	GUI	An integer from 0 – number of blue pugs detected
<i>lost</i>	nucleo	GUI	An integer from 0 – number of lost pugs
<i>unpainted</i>	nucleo	GUI	An integer from 0 – number of unpainted pugs detected
<i>bufferspeed</i>	nucleo	GUI	Integer from -100 to 100. Speed of the buffer wheel
<i>tubes</i>	nucleo	GUI	An integer from 0 – 3 number of filled sorting hoppers
<i>recipe</i>	nucleo	GUI	An integer from 1 – 11 each number represents a specific colour recipe
<i>stage</i>	nucleo	GUI	An integer from 0 – 6. Represent the current stage in the cycle.

- **Statistics tab:**

When the livestream in the home page/tab is done, the data is stored on the statistics tab. All topics here start with Kelvin/PugsGo/stats/...

<i>Topic</i>	<i>Publisher</i>	<i>Subscriber</i>	<i>Payload</i>
<i>loaded</i>	nucleo	GUI	An integer from 0 (loaded) pugs
<i>painted</i>	nucleo	GUI	An integer from 0 (painted) pugs
<i>sorted</i>	nucleo	GUI	An integer from 0 (sorted) pugs
<i>red</i>	nucleo	GUI	An integer from 0 – number of red pugs detected
<i>green</i>	nucleo	GUI	An integer from 0 – number of green pugs detected
<i>blue</i>	nucleo	GUI	An integer from 0 – number of blue pugs detected
<i>lost</i>	nucleo	GUI	An integer from 0 – number of lost pugs
<i>unpainted</i>	nucleo	GUI	An integer from 0 – number of unpainted pugs detected
<i>stage</i>	nucleo	GUI	An integer from 0 – 6. Represent the current stage in the cycle.

Then, every time a recipe is selected, it will also be published to the GUI as:

Kelvin/PugsGo/stats/ (recipe initials e.g., rgb). For example, the recipe called bi\_rb => Kelvin/PugsGo/stats/rb.

### 3.2. GUI to NUCLEO

The GUI only publishes 5 topics to the NUCLEO. This is because the MQTT library had a limit of 5 message handlers. According to research, this setting can be increased in the MQTTClient.h file but When I changed it, I still could not increase my subscriptions. Therefore, I had to strip away a few other settings from my settings page so that I could only facilitate the most important ones All topics on the settings page have the prefix, Kelvin/PugsGo/set/...

<i>Topic</i>	<i>Publisher</i>	<i>Subscriber</i>	<i>Payload</i>
<i>set/tube_size</i>	GUI	nucleo	An integer from 0 -10. Number of pugs needed for a pug to be called full
<i>set/crecipe</i>	GUI	nucleo	An integer from 1-11. Each number represents a specific colour recipe
<i>set/tubes</i>	GUI	nucleo	An integer from 0 – 3. Number of sorting hoppers that should be filled
<i>pause</i>	GUI	nucleo	1 - represents the clicking of a pause button
<i>continue</i>	GUI	nucleo	1 - represents the clicking of a continue button

Note also that the “pause” and “continue” topics are not on the settings tab but on the home tab.

#### 4. Mobile device GUI and functionality

The rest of the project was the Graphical user interface. I wanted to create a GUI that not only allows control and feedback between the user and the microcontroller but also something that produces useful information and insights by generating data that can be analyzed to help the user to make informed decisions.

Note that in this section different layouts are used for demonstration this is because I have been continuously trying to improve the user interface.

The GUI is divided into 3 tabs. The home tab, the stats tab and the settings tab.

##### 4.1. Home Tab

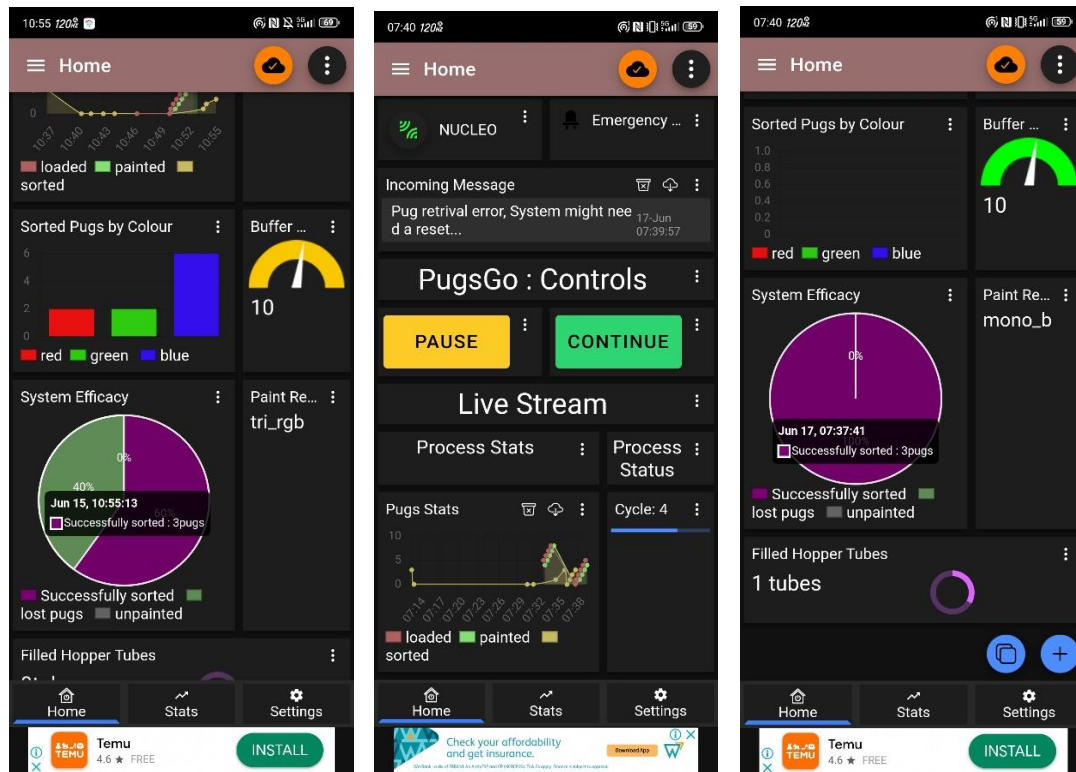


Figure 17: Home Page/Tab

The home page/tab allows the user to visualize what is going on in the simulation. Every pug loaded, every pug painted, lost, unpainted or sorted is accounted for by use of various charts. The user also can pause and continue the system online and also get feedback if the operation went through.



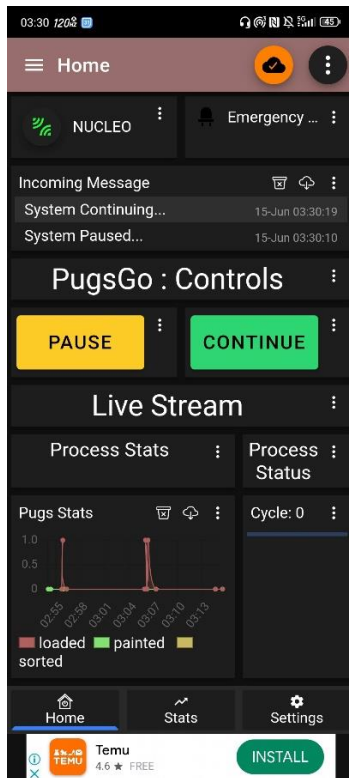


Figure 18: System paused

Given that MQTT is cloud based and the user could be operating from anywhere in the world, it was not ideal to give them access to functionalities like emergency stop or even starting the system. These tasks require someone on the ground. In case there is an emergency and someone on the ground has triggered the Estop, the led on the home tab starts to flash and all other operations are paralyzed in this case.

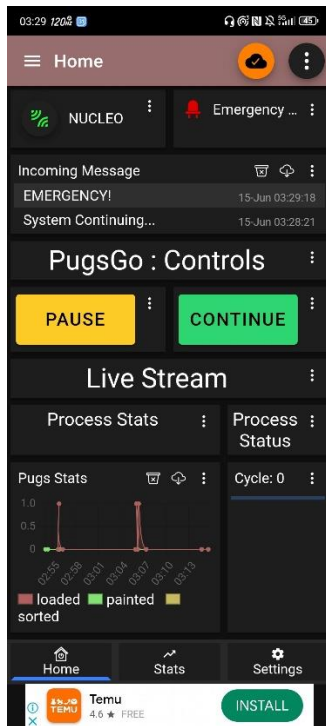


Figure 19: Emergency alert

The user is also able to see the status of the system like the current set speed of the buffer wheel, the current set cooler recipe, the number of hoper tubes that have been filled so far, and the stage of the current cycle. If the buffer wheel turn red it means that with the current speed most tasks on the simulation will not be stable.

The cycle stages are interpreted thus:

- 1: pug is being retrieved by the hopper cylinder
- 2: pug is in the loading area
- 3: buffer wheel is now moving towards the paint gun (ideally with a loaded pug)
- 4: a pug is being painted
- 5: the pug unloading phase
- 6: the pug sorting phase

Once a cycle (of 8 pug) is over all the numerical data on the home page is cleared (the livestream is over) to give way for the next cycle. All the data since the machine started running is however kept on the stats (statistics) page.

Also note that when you touch on a section of a pie chart or a point on a graph, it displays further details.

## 4.2. Stats Tab

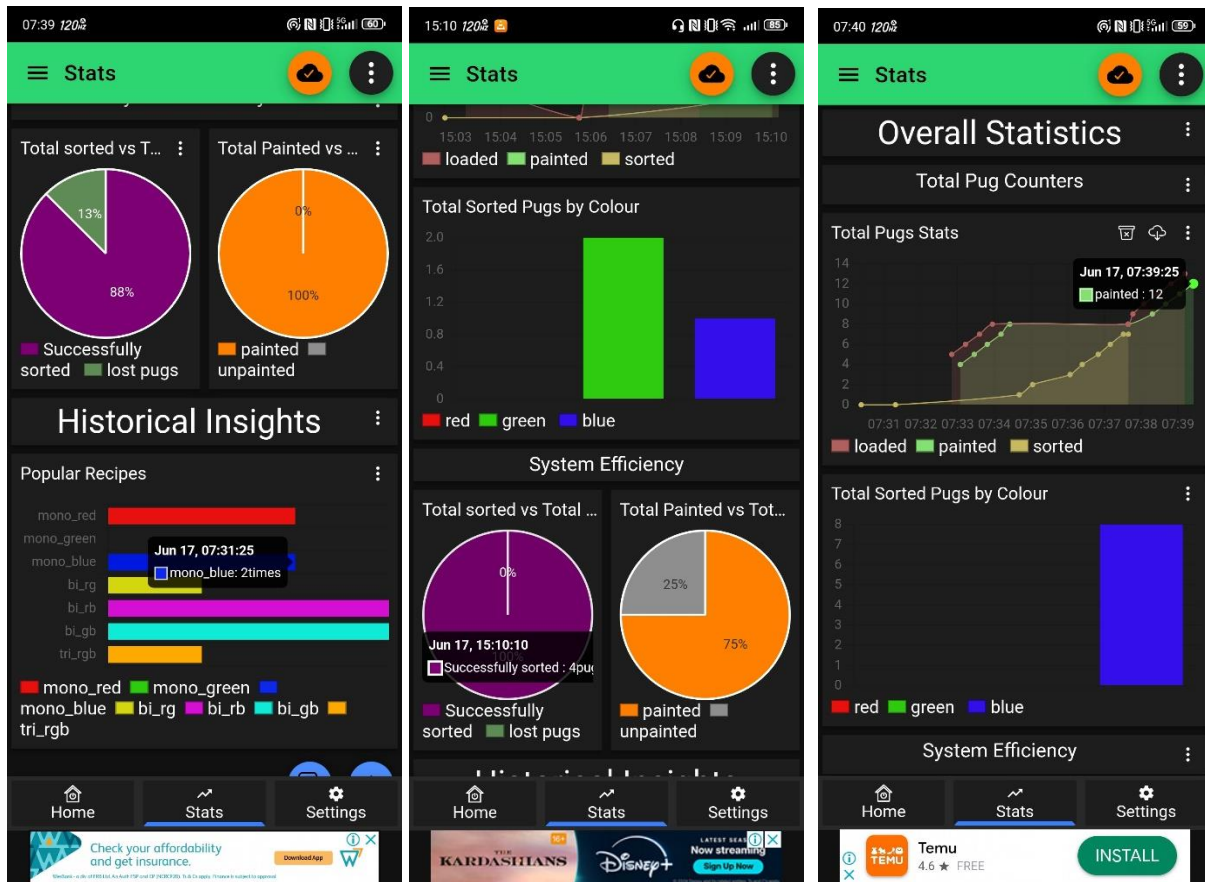


Figure 20: Stats page

This is the tab that provides an overview of how the process is going.

### 4.3. Settings Tab

This tab allow the user to change parameters in the NUCLEO source code while online and alter the operation of the simulation.

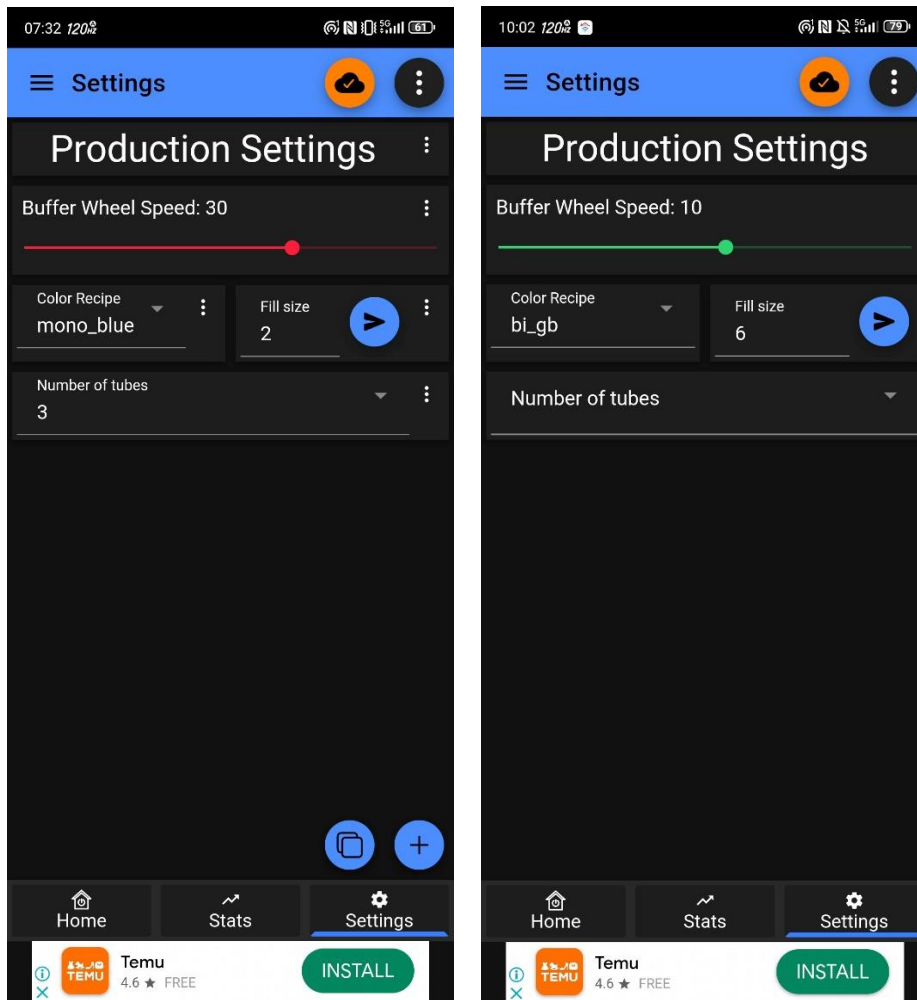


Figure 21: The settings page

The buffer speed panel has been deactivated (the nucleo does not handle this topic) to give room for more important parameters.

## 5. Operating Manual

### 5.1. Useful Tips

- **Fill size:** By default (with or without the GUI connected), the fill size is set to 3. This is the number of pugs required to fill a sorting hopper tube. Once a tube has received that number of pugs, it becomes full. This number ranges from 1 to 10. Beyond this range, the default value will be applied.
- **Number of tubes:** By default, this value is set to 3. This is the number of sorting hoppers that must be filled in the current session. Once each tube has got 'fill size' number of pugs, the session is over. The system must be reset to start another session. This value ranges from 1 to 3.

- **Recipe Names:** The 'Colour Recipe' panel has got coded names for recipes.
  - **mono-(X):** these are single colour recipes. All pugs have the same colour.
    - ◆ To create a R-R-R sequence in all tubes for example, the color recipe should be set to mono\_r, Fill size to 3 and Number of tubes 3 as well.
    - ◆ To create G-G-G-G-G, the color recipe is mono\_g and fill size is 5.
  - **bi-(XY):** these are 2 colour recipes.
    - Possible sorting: RG-RG-RG, in this case we would set the fill size to 6 and colour recipe to bi\_rg. And then adjust Number of tubes to the number of sorting hoppers we want to fill. If not adjusted, the default is 3 (all hoppers). If fill size is 2 all the tubes end up containing 2 RG pugs each.
  - **tri-(XYZ):** these are 3 colour recipes and the sequence goes like RGB-RGB-RGB. If fill size is 6 each tube ends up with 2 sets of RGB pugs.

It is ideal that for 2 colour recipes, the fill size should be a multiple of two and for 3 colour recipes, fill size should be a multiple of 3. This way all the tubes will achieve uniformity.

## 5.2. How To...

- Inside the source code (NUCLEO code) use the shortcut keys (Ctrl + F) and search for "NSAPI\_SECURITY\_WPA2". This will take you where you need to adjust the WIFI credentials.
- When system turns on and all the 8 Leds on the SWLed shield are on, the system is in production mode. Use SW7 on the shield to toggle between Test mode and Production mode.
- Wait for NUCLEO to establish MQTT connection with the broker. Feedback is printed on the simulation console.
- If you want to adjust the recipe parameters, do it before pressing the start button.
- In the GUI app, navigate to the 'Settings' tab to adjust parameters. The tab navigation buttons are at the bottom of the screen and have got icons to assist the user.
- Press the start button to start the sorting process.
- If a catastrophic error occurs, reset the system and start over again.
- If you change the recipe while system is running, it will only take effect after the current task is complete.
- While the system runs keep an eye on the 'Incoming message' panel in the home tab for communications from the GUI. The panel however keeps a few messages before deleting them in case the user misses something.
- Navigate between the home tab and the stats tab to see how the system is performing.
- The home page gives a livestream experience while the stats tab keeps all the data even after many cycles of the system running.

- Watch out for pug errors that might compromise the recipe.