Automated Plant Care

1 Introduction

The goal of this project is to implement reliable automation for plant care. Taking care of plants that yield something of value to the caretaker or his or her employer boils down to supplying the proper levels of light, hydration, and nutrients. Of these, the first two typically require the caretaker's attention at least once per day. The most severe potential consequence of neglect is, of course, death of the plant, which renders the entire effort futile.

The situation at hand involves a chronically negligent plant caretaker coupled with a low-light environment due to a large bush obstructing the only window. What is documented here are the steps involved to establish a back-end for the automation itself that is paired with an informative front-end that also sends alerts for key events, e.g. a malfunctioning pump.

2 System Overview

The two fundamental pieces of equipment are the light and the pump, and what they both have in common is that they run on AC power. Obviously, the controller for all of this – a Raspberry Pi – is only equipped to handle DC circuits at either 3.3 or 5 volts. While it's possible to do something with relays, nobody wants to explain their neat project to an angry fire marshal. Thus, for safety reasons, it has been elected to use a radio frequency (RF) transmitter paired with RF outlet switches to enable switching of the pump and light. This has an added benefit of greater freedom for routing power. The remainder of equipment to be used is given in the table below, and Fig. 5 provides the *initial* schematic of the design.

Table 1: Equipment specifications for plant care and monitoring.

Item	**Purpose**	**Seller or Spec**
AC outlet switches	Allow RF control of AC appliances	Amazon
Hygrometer	Monitoring of soil moisture	AliExpress
Pump	Supply water to plant	Amazon
Light	Provide correct spectrum for plant	Erligpowht 45 W grow light
Photoresistor	Ensure light is operating	GM5539
Raspberry Pi	Runs at least the back-end	Raspberry Pi 3B
Raspberry Pi camera	Visual of plant and maybe time-lapse	V2
Arduino Uno	Report analog sensor data	Amazon

2.1 Operation

The most trivial part of how the system will operate pertains to the light, which will run on a 12 hour on/off schedule. Half-way through the day portion, the Raspberry Pi will take a still image of the plant. The photoresistor will be used to ensure the light is working as it should.

Next up is the pump operation. When the hygrometer indicates dry soil, the pump will operate for a brief TBD interval. This pump will be submerged in a reserve of water that will have a simple circuit that's nominally closed by the water itself until the reserve is nearing depletion.

2.2 Front-End Requirements

The front-end will provide a simple, vertically scrolling interface with metrics about delivery of light and water to the plant in the form of two time-series plots for the last 24 hours. Vertical axes will be the light level from the photoresistor and the moisture level from the hygrometer, respectively.

Below the plots, there will be status indicators for the light level, moisture level, and water reserve level. These will be

boolean in nature, and the plant care-taker shall be notified in some way when one of these strays from nominal. At the bottom of the interface, the most recent picture of the plant will be displayed.

Hosting of the front-end will be on an AWS instance running an AMI generated by Packer and provisioned via Terraform. On this instance, a web server and a database will run in Docker containers orchestrated by Docker Compose. The web page will be open to the web, and everything else will only be open to the public IP of where the Pi is located.

3 Sensors and Circuits

3.1 RF Outlets

The starting point is to establish RF capability between the Pi and the RF outlets, which was accomplished by following the steps put forth by GitHub user timeland's blog post and the associated repository rfoutlet. The basic procedure is to use an RF receiver to decode signals from the remote that came with the outlets, then use the RF transmitter to emulate the signal for a given channel. This yielded the codes given in the table below, and Fig. 1 shows the setup used. Also, since it's not explicitly called out, the transmitter and receiver use wiring pi pins zero and two, respectively.

Table 2: Codes transmitted by RF outlet remote.

Channel	On	Off
1	1398067	1398076
2	1398211	1398220
3	1398531	1398540
4	1400067	1400076
5	1406211	1406220

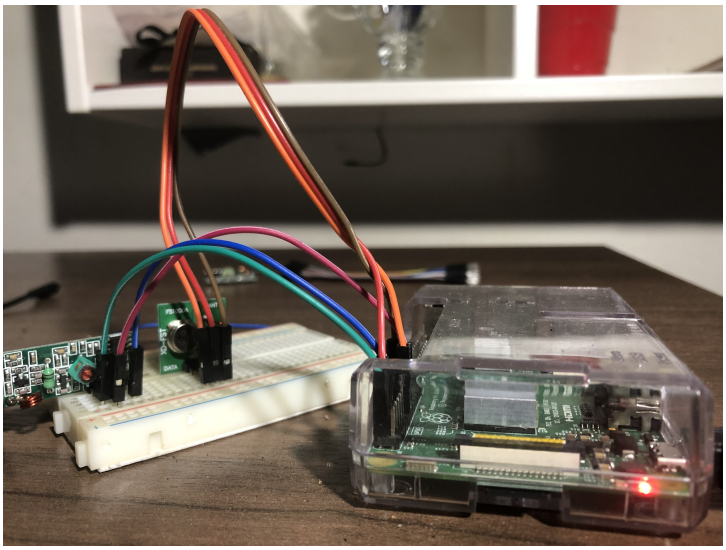
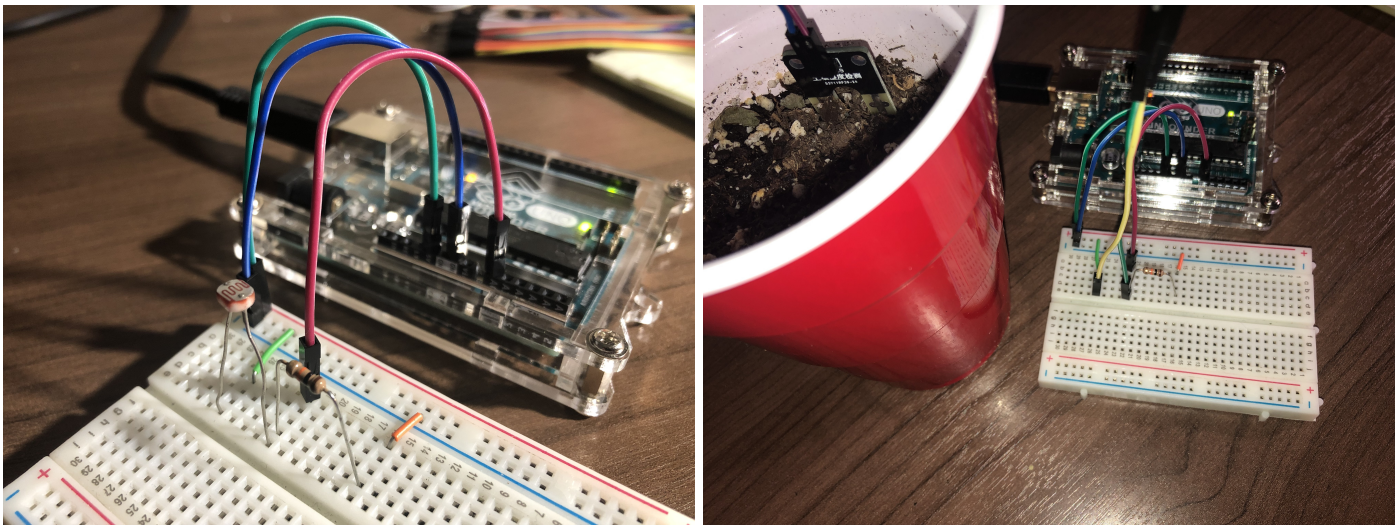


Figure 1: Setup for getting RF codes.

3.2 Photoresistor

A key detail is that a photoresistor is an analog device. Solutions for the Pi tended to involve capacitors that were not immediately available, so it was decided to use an Arduino Uno to handle analog inputs. Viewable in Fig. 2a, the circuit for the photoresistor is a simple voltage divider, with one side wired to pin A0 and the other being a 10k Ω resistor leading to ground.

From here, using the Arduino IDE's serial monitor/plotter, it was verified that this arrangement yielded higher values with more light. It is worth noting that shadows cast substantially affect readings, so the sensor will have to be located in a place that won't be obstructed by leaves.



(a) Photoresistor circuit.

(b) Hygrometer circuit.

Figure 2: Test circuits for analog sensors.

3.3 Hygrometer

The hygrometers purchased on Ali Express came with a small chip containing a potentiometer that converts the analog signal to a binary, digital one. The intended use is to set the potentiometer to a threshold value that would serve as a set-point for watering.

For this case, a low/high sensor isn't terribly interesting, so per Fig. 2b, the photoresistor was swapped with the hygrometer in a cup of dry soil. To test the sensor, water was poured into the cup. Fig. 3 shows the serial plotter output for pouring a small bit of water followed by a sustained pour. By this metric, it seems feasible to incorporate the hygrometer as an analog device, which enables more interesting plots versus time than the digital option.

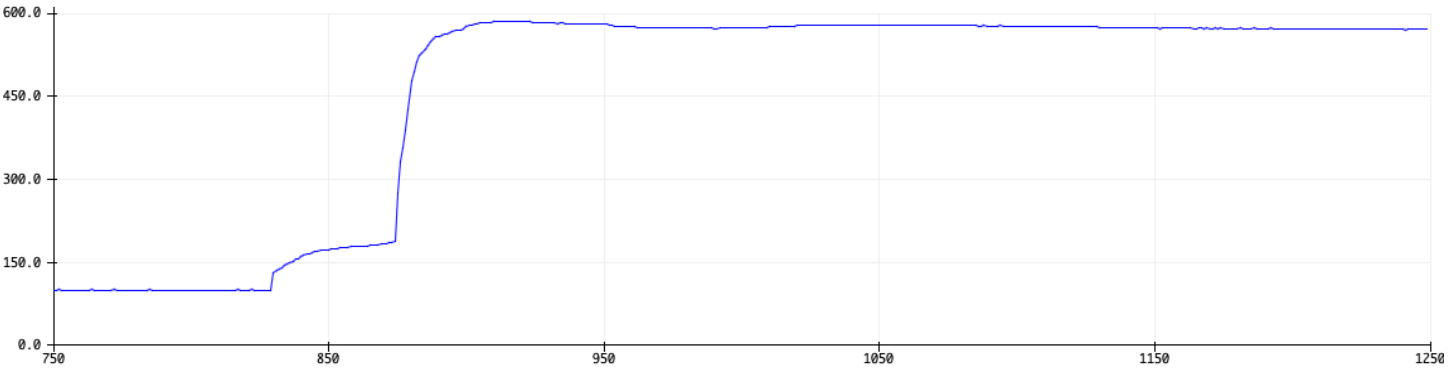


Figure 3: Soil moisture versus time (arbitrary units) as water was poured into the cup.

3.4 Water Level

This is certainly the easiest of the sensors to describe and implement. Similar to the others, a voltage divider will be used. When the leads are submerged in the water, the circuit is closed with negligible resistance, so the only change to the general circuit in use is to have another 10k Ω resistor placed in series with the analog pin to avoid a short. Given the trivial nature of the sensor, it was not deemed necessary to test this circuit separate from the others.

3.5 Summary and Combined Operation

Investigating use of a photoresistor, hygrometer, and a simple water level sensor led to a few key findings. The most significant is that an Arduino is needed to bridge the gap between analog and the Raspberry Pi. Along with this, it was found that the same basic circuit – a voltage divider with a 10k Ω resistor on ground – can be employed for all three sensors.

These criteria informed the writing of the sketch shown in Fig. 6, which simply prints the state of the three pins delimited by commas. It is thought best to keep the Arduino sketch dead simple, because there's not a lot to work with as far as libraries or computing power. Pin mapping is given in the table below.

Table 3: Arduino pin mapping.

Pin	Sensor
A0	Photoresistor
A1	Hygrometer
A2	Reserve Water Level

From here, the Raspberry Pi can fetch sensor data as needed over a USB connection. Fig. 7 shows the Python script that was used to test this combined functionality, and the physical setup given in Fig. 4 was used to successfully test analog sensing and RF outlet switching.

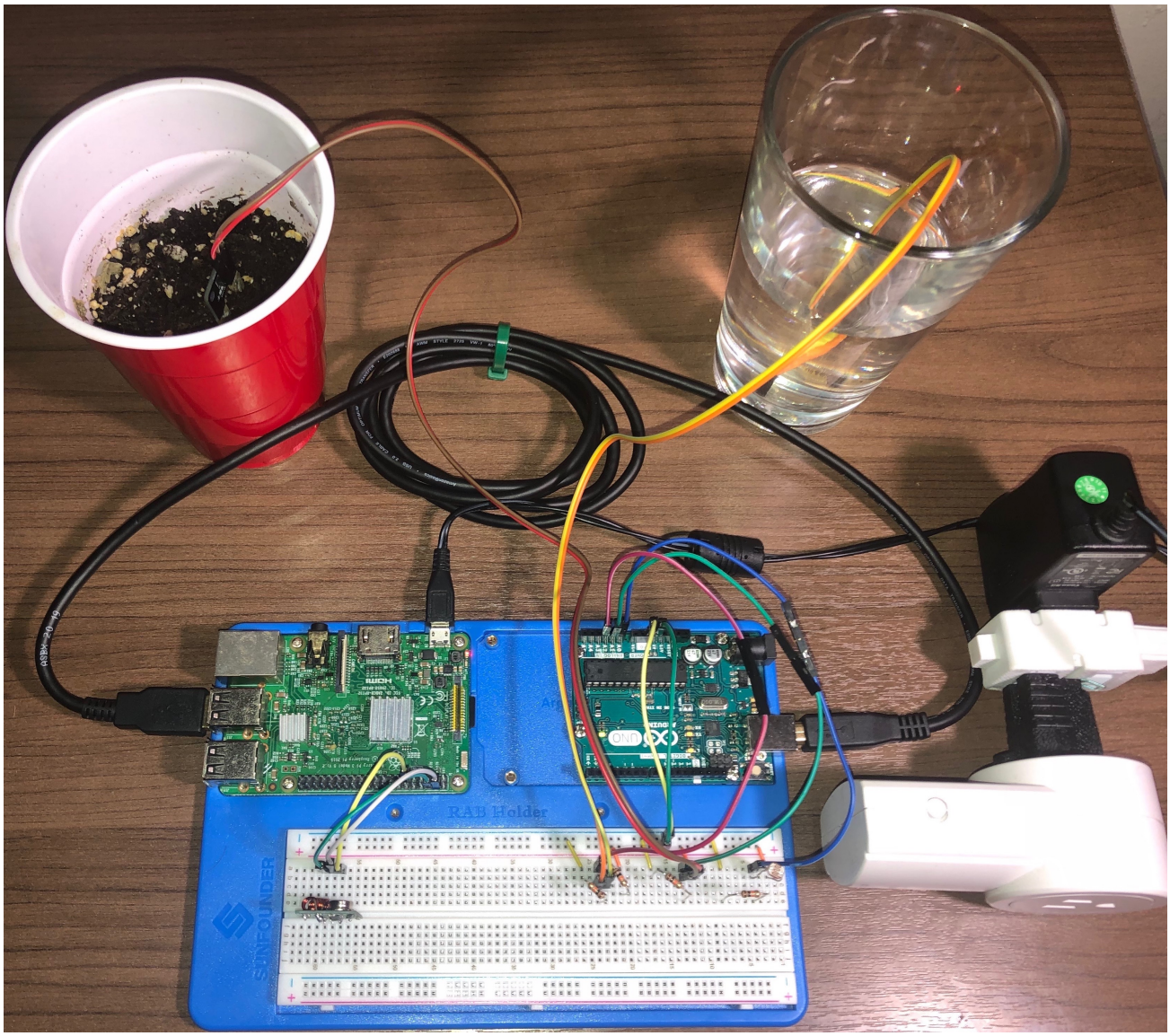


Figure 4: Combined test setup for analog sensing and RF outlet switching.

4 Software Design

For any project, it's best to start with, considering the required inputs and outputs for, in this case, the Raspberry Pi. Inputs are sensor data from the Arduino, the current time, and camera stills. Local outputs are RF outlets for the light and pump. Outputs to a remote host include historic sensor data and the latest image from the camera.

Given what's known so far, it seems safe to say that this is an event-driven program. The table below summarizes the anticipated events and the envisioned response from the program.

Table 4: Summary table of event logic.

Event	**Response**
Light not detected when "on"	Light status to error
Soil dry after pump run	Pump status to error
Reserve level reads low	Warn low reserve level
Dry soil	Run pump for interval
Local time 8 am	Turn on light
Local time 2 pm	Take still image
Local time 8 pm	Turn off light

4.1 Tooling

A significant design question is the choice of language – either Python or C++ with Boost. The biggest pro for Python is a much lower barrier to entry per Fig. 7, but what's most appealing about C++ is the option to integrate new code with what's in place for controlling the RF outlets. Given that and Fig. 6, this choice would also mean that the entire project is written in the same language, and just a single program would have to be run on the Pi.

This tips the scales in favor of C++ with Boost in spite of the added nuance. Libraries exist for serial communication, and the camera can be accessed using the built-in executable `raspistill`. To take advantage of new features in `std::filesystem` and similar, C++ 17 has been elected as the project standard.

Table 5: Summary of C++ tooling on the Raspberry Pi.

Tool	**Version**
make	4.2.1
g++	8.3.0
gdb	8.2.1
CMake	3.15.2
Boost	1.67

4.2 Implementation

Since Raspbian ships with the executable `raspistill`, there's no sense in writing redundant code, so pictures are obtained by invoking `raspistill` via `std::system`. The actual starting point, however, was the Boost serial library followed by a search for examples on how to use it. Implemented types are as follows.

SimpleSerial This type is a wrapper of what's needed to read serial input asynchronously. The example this was drawn upon had a reasonable implementation for reading a single line, so a parsing function to convert the string into a vector containing the three expected integers was added.

OutletController Methods contained in this type provide an interface to turn the light on or off as well as cycling the pump for a specific number of seconds. The syntax for actually transmitting a code is wrapped up in `send_code`.

PlantWatcher Per its name, this type is responsible for monitoring and orchestrating the respective events and responses detailed in Table 4 by interfacing with `SimpleSerial` via `update_sensor_data()` and by having a pet instance of `OutletController` to operate on. Last of all, the method `update_photo()` ensures that a still image is captured and stored a the appropriate time.

5 Front-End Implementation

6 Appendix

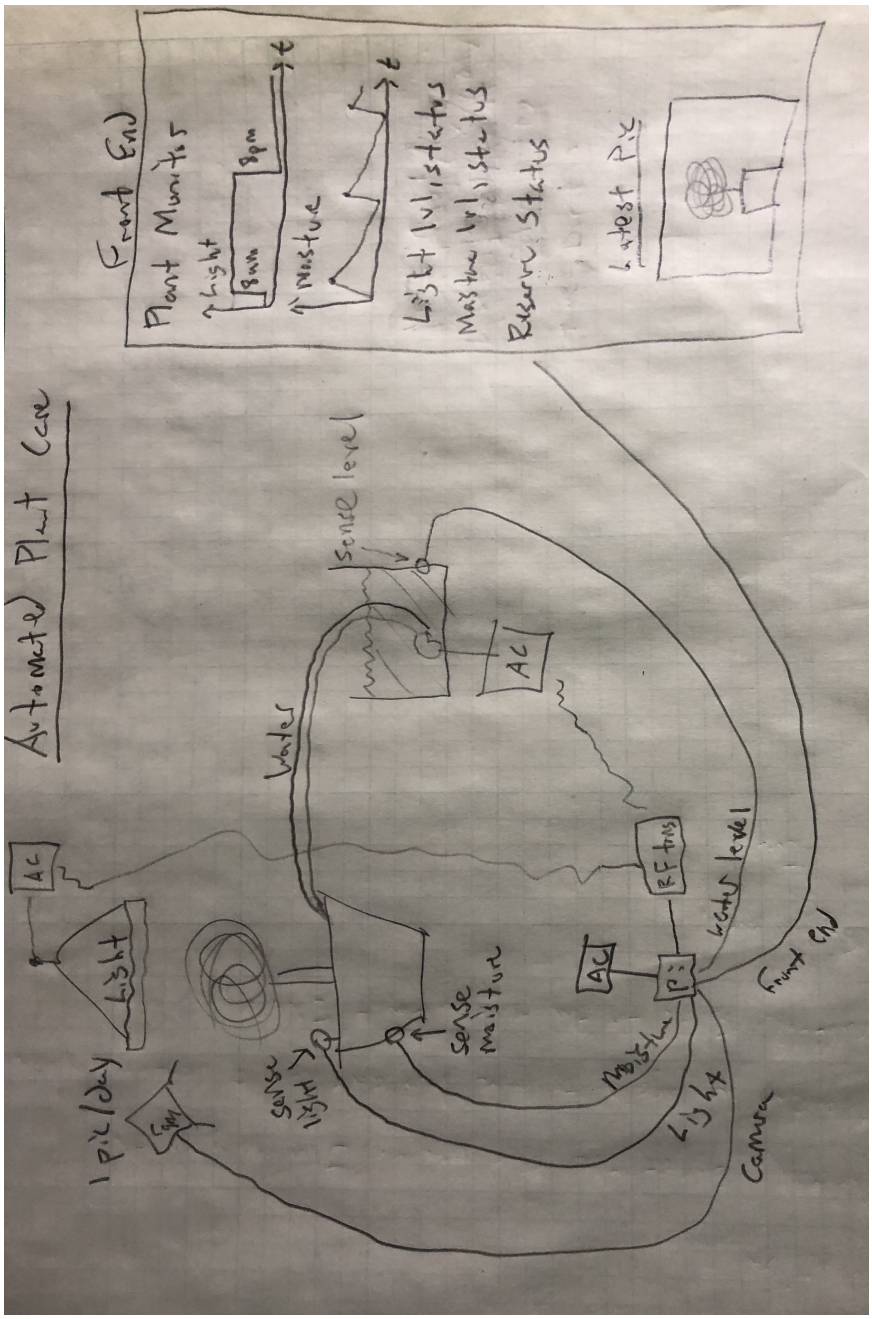


Figure 5: Initial schematic of plant automation system.

6.1 Serial I/O Code

```
/*
 * This is a simple sketch for polling analog sensors used to track
 * levels of water and light being supplied to the plant.
 *
 * Pin layout
 * ----------------------
 * A0: Photoresistor
 * A1: Hygrometer
 * A2: Water level sensor
 * ----------------------
 */

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  for (int i = A0; i < A3; i++)
  {
    // Read and map to 8 bits
    int val = analogRead(i);
    val = map(val, 0, 1023, 0, 255);

    // Print val
    Serial.print(val);

    // Print comma to coninue line or newline
    char tailChar = ',';
    if (i == A2)
    {
      tailChar = '\n';
    }
    Serial.print(tailChar);
  }

  delay(50); // Milliseconds of pause between reads
}
```

Figure 6: Arduino sketch for reading the photoresistor, hygrometer, and water level sensor.

```
import serial
ser = serial.Serial('/dev/ttyACM0', 9600)


def line_to_list(line):
    """ Parses line string into a list of three ints."""
    good_line = True
    req_len = 3

    try:
        # Remove whitespace and split by comma
        line = line.strip()
        line = [int(x) for x in line.split(',')]

        # ValueError will be thrown for non-numbers or wrong length
        if not len(line) == req_len:
            msg = "ERROR: Expect {} ints.  Got {}".format(req_len, line)
            raise ValueError(msg)
    except ValueError:
        good_line = False
    finally:
        # Need validity of the line and the parsed list
        return [good_line, line]

# Simple format to print what each value corresponds to
msg_str = "Light: {}, Moisture {}, Reserve Level: {}"

# Print Arduino sensor data indefinitely (till stopped)
while 1:
    if(ser.in_waiting > 0):
        line = ser.readline()
        [good_line, line] = line_to_list(line)
        if good_line:
            print(msg_str.format(*line))
```

Figure 7: Simple script to test parsing Arduino sensor data on the Raspberry Pi.

6.2 Exemplary Provisioning Code

```
{
  "builders": [{
      "type": "amazon-ebs",
      "access_key": "{{user `aws_access_key`}}",
      "secret_key": "{{user `aws_secret_key`}}",
      "region": "us-west-1",
      "source_ami": "ami-056ee704806822732",
      "instance_type": "t2.micro",
      "ssh_username": "ec2-user",
      "ami_name": "amazon-docker {{timestamp}}"
    }
  ],

  "provisioners": [{
      "type": "file",
      "source": "./install_docker.sh",
      "destination": "/home/ec2-user/install_docker.sh"
    },
    {
      "type": "shell",
      "inline": [
        "chmod +x *.sh",
        "sudo ./install_docker.sh"
      ]
    }
  ]
}
```

Figure 8: Packer syntax used to generate an AMI with Docker installed.

```
provider "aws" {
  profile    = "default"
  region     = "us-west-1"
}

resource "aws_instance" "front-end-server" {

  // Use the AMI from Packer
  ami           = "ami-0d3ea0b56fa289e40"

  // Want to be in the free tier
  instance_type = "t2.micro"

  // Want ingress from SSH, Redis and egress to open net
  vpc_security_group_ids = ["${aws_security_group.plant-sg.id}"]
  key_name = "plant_fe"

  // Name for display in console
  tags = {
      Name = "Plant-Monitor-Frontend"
  }
}

// Want a known, static IP for the remote
resource "aws_eip_association" "eip_assoc" {
  instance_id   = "${aws_instance.front-end-server.id}"
  allocation_id = var.eip-id
}
```

Figure 9: Provisioning syntax for an AWS instance with an elastic IP.

```
resource "aws_security_group" "plant-sg" {
    name = "plant-sg"
    description = "Set ingress/egress for front-end server"
    vpc_id = var.vpc-id

    // Incoming SSH from local IP
    ingress {
        from_port   = var.port-ssh
        to_port     = var.port-ssh
        protocol    = "tcp"
        cidr_blocks = ["${var.local-ip}/32"]
    }

    // Incoming Redis client from local IP
    ingress {
        from_port   = var.port-redis
        to_port     = var.port-redis
        protocol    = "tcp"
        cidr_blocks = ["${var.local-ip}/32"]
    }

    // Outgoing to open net
    egress {
        from_port   = 0
        to_port     = 0
        protocol    = "-1"
        cidr_blocks = ["0.0.0.0/0"]
    }
}
```

Figure 10: Security group configuration.