Team Null

Detailed Design

Version 1.0

Revision History:

- 0.1 Created doc added section headers- Lawrence Onyango-12/8/2023
- 0.2 Created localization section and conducted subsequent trade studies - 12/8/2023
- 0.3Created design details - Jai Kumar 12/8/2023
- 0.4 Created fault recovery methods - Ethan Todd 12/8/2023
- 0.5 Created degraded operational modes - Ethan Todd 12/8/2023
- 0.6 Created the software architecture and safety section - Harshul 12/8/23
- 0.7 Refractored document; Added project description, subsystem decomposition, created parts list - Lawrence Onyango 12/17/2023
- 0.8 Refined design CAD, and descriptions for each design subsystem - Jai 12/17/2023
- 0.9 Refined and updated software architecture - Harshul 12/18/2023
- 1.0 Created and added section on temporal state with state charts to show the general operation in 2 scenarios. - Harshul 12/19/23

Project Description

At Team Null, we envision creating an autonomous garden maintenance robot that will help hobbyists and other users maintain their gardens. This innovative solution integrates computer vision, LIDAR, and SLAM techniques to autonomously navigate gardens, meticulously analyzing individual plants using a soil sensor probe. The robot would then analyze the soil readings to precisely administer water and nutrients, adjusting care methods in real-time. Through rigorous validation and prototyping, we aim to deliver a self-sustaining, intelligent gardener, revolutionizing horticulture and agriculture practices.

Sub-System Descriptions

Functional Decomposition:

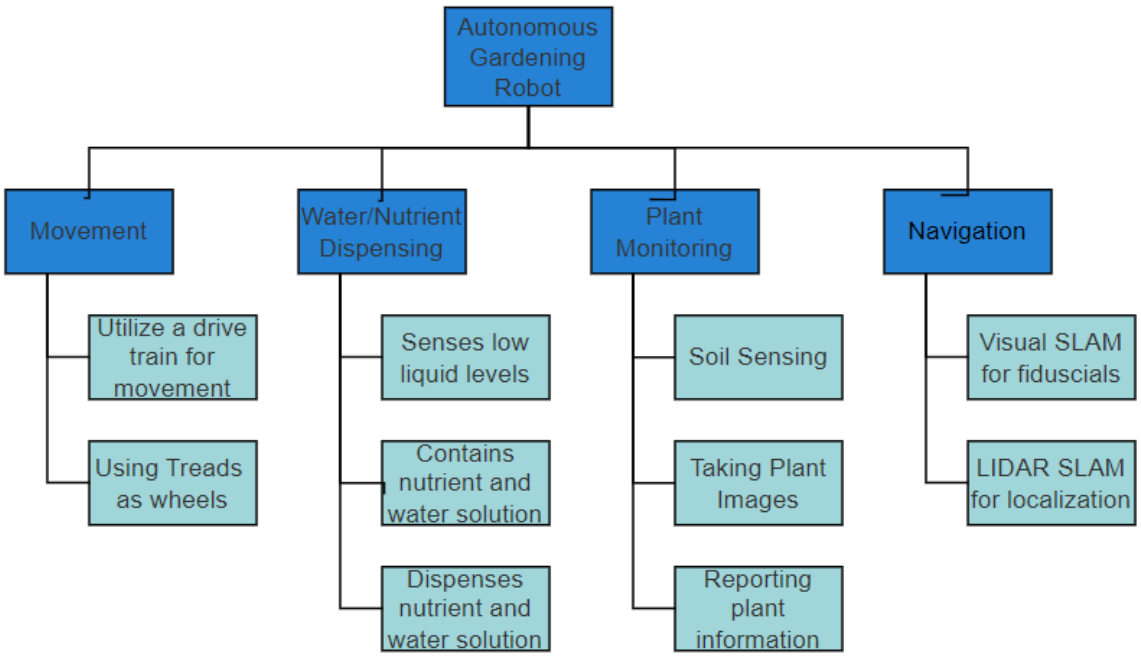


Figure 1: *A diagram showcasing our functional decomposition tree, highlighting our main four subsystems: Movement, Water/Nutrient Dispensing, Plant Monitoring, and Navigation*

Movement:

The movement subsystem will be responsible for propelling the gardening robot across various terrains within the garden environment. It will incorporate a tread-style drivetrain powered by dual 12V DC motors, offering significant torque and maneuverability over different surfaces. The robot's design includes a wheelbase of 15 inches width and 12 inches length, ensuring it can navigate between rows of plants efficiently. Equipped with fault recovery mechanisms, the movement subsystem will maintain the robot's stability and safety by promptly identifying and addressing any actuator irregularities or faults, ensuring smooth and uninterrupted operation while preserving the integrity of the garden.

Water and Nutrient Dispensing:

This subsystem oversees the precise administration of water and nutrients critical for plant care. It comprises dual nozzles strategically positioned on either side of the robot, this system harnesses custom-designed reservoirs with a capacity of around 500mL each. These reservoirs feature specialized inlets for easy refilling—one for water and nutrient solutions and another for pressurizing the liquid to facilitate controlled dispersal through the nozzles. Governed by solenoids, these nozzles ensure accurate and targeted spraying, guaranteeing optimal plant hydration and nourishment while averting excessive watering or nutrient application.

Plant Monitoring:

The plant monitoring subsystem serves as the visual and soil-sensing component of the autonomous gardening robot. It consists of a camera that captures detailed images of plants, enabling the analysis of their health and growth patterns using computer vision techniques. Simultaneously, the system integrates a soil sensor probe that measures soil moisture content and pH levels. This combined approach provides comprehensive data on plant condition and soil health, allowing the robot to make precise adjustments in watering and nutrient distribution for optimal plant care. The data collected by this subsystem will be transferred from the robot to the user in intervals.

Navigation:

The navigation subsystem will play a pivotal role in facilitating the autonomous movement of the gardening robot within the garden environment. It will utilize a combination of Visual-Based Simultaneous Localization and Mapping (SLAM) alongside LiDAR SLAM techniques to accurately determine its position and map its surroundings in real-time. By leveraging ROS SLAM toolbox and associated open-source software libraries, it will process visual data from cameras and depth information from LiDAR sensors. This integration will enable the robot to effectively maneuver through the garden while avoiding obstacles, ensuring precise plant care and efficient garden maintenance.

Detailed Design

With our individual subsystems defined, we created the following CAD model to reflect our chosen subsystem concepts and overall system requirements. The following figure depicts our CAD with each subsystem highlighted:

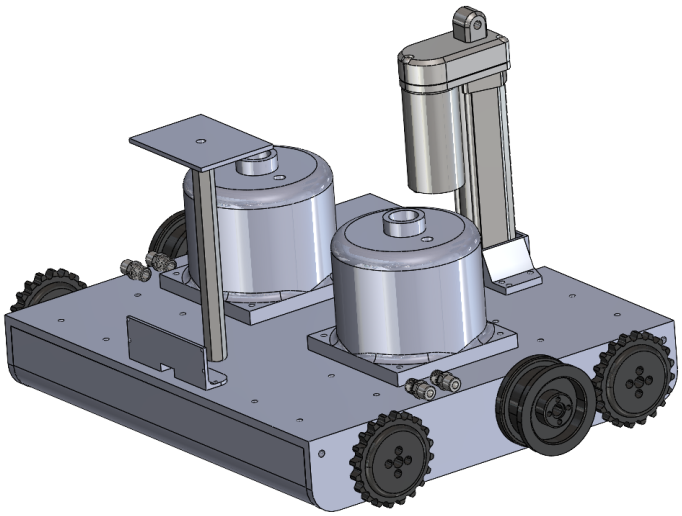


Figure 2: *Isometric view of Overall Assembly (CAD)*

Drivetrain:

To ensure that our robot is capable of traversing over most terrains, we have elected to use a tread-style drivetrain as opposed to wheels (treads not depicted in CAD model, driving sprockets are shown). The tread belts will be tensioned with idler wheels, which will have a 10mm slit to be adjusted. If the tread belt is still loose, then a link can be removed.

Each side of the drivetrain will be driven by two 12V DC motors which output a maximum of 100 rpm and 700 oz. in. of torque. This will give us ample power to support the whole weight of the robot chassis and payload and be able to drive at a high enough speed within the garden. As the speed of plant growth is relatively slow, the speed of the robot does not need to be as high. If we find that the motors are struggling under the weight of the robot (which is unlikely to happen), the motors have swappable gearboxes such that we can increase the maximum torque.

The total wheelbase spans a width of 15” and a length of 12”. A wider wheelbase would also allow our robot to have increased stability and maneuverability when traversing the garden. This especially reduces the risk of tipping over, which may be caused by slashing liquids in the robot’s reservoir tanks. With the specifications of the garden (specifically a column width of 2’ between plants), we are confident that the robot will be able to perform without interfering with the garden.

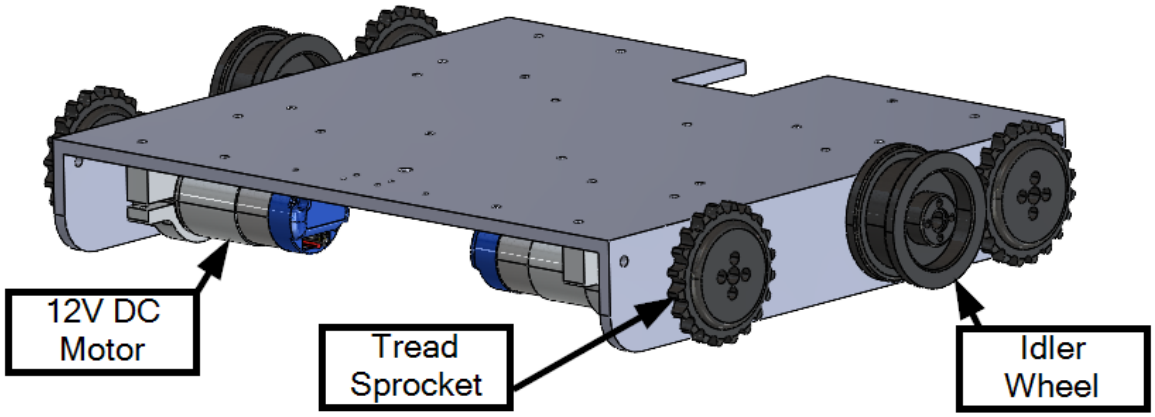


Figure 3: *Sprocket drive for tread based drive-train*

Water and Nutrient Dispensing Subsystem:

The Water and Nutrient Dispensing subsystem will efficiently manage the distribution of essential elements crucial for plant growth and health within the garden. It incorporates a mechanism consisting of two nozzles on each side of the robot, specifically designed for dispensing water and nutrients to the plants. These nozzles are fed by custom 3D-printed reservoirs, each with a capacity of approximately 500mL, and equipped with two inlets. One inlet enables a user to pour in water and nutrient solutions, while the other contains a valve to pump air in. We will be using the McMaster 60-degree cone nozzle to ensure that the entirety of the plant and soil area is covered. Beneath these reservoirs, a force sensor is integrated to accurately gauge the liquid levels. We will calibrate the force sensor based on the weight of the reservoir tanks to allow us to measure the actual volume of liquid in the tanks. This will allow us to detect when the robot is low on water and nutrients, requiring the robot to go back to its home base.

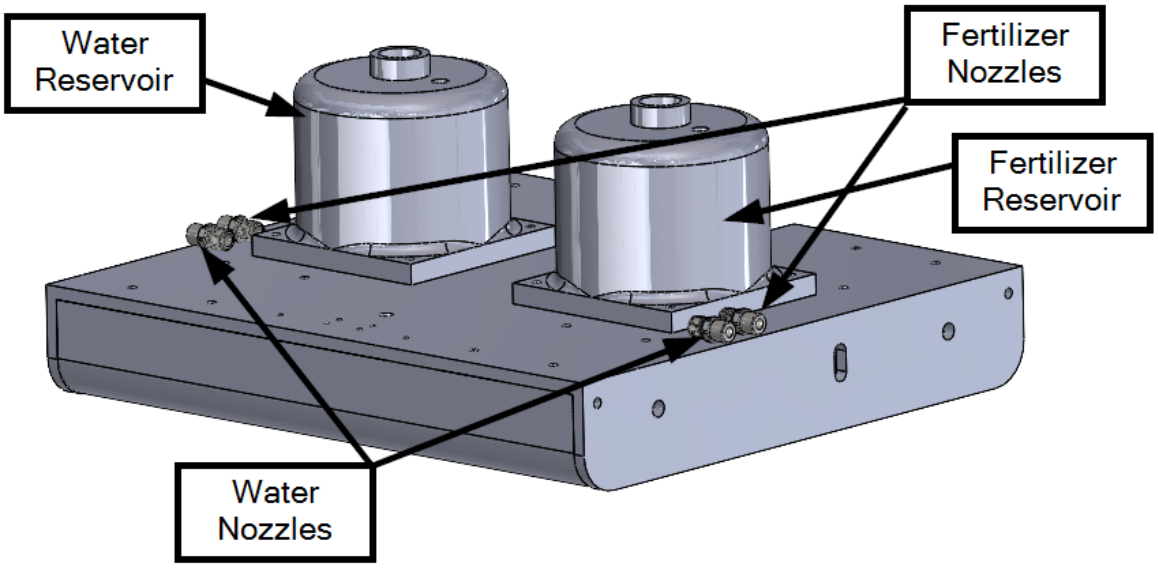


Figure 4: *Reservoirs and nozzles on the chassis.*

Plant Monitoring Subsystem:

The Plant Monitoring System will be using a robust steel-probe soil sensor to measure the soil moisture and pH values. A Firgelli Automations' Linear Actuator (3” stroke, 35lb force) will be used to insert the steel probes into the ground, ensuring that we can reach a sufficient soil penetration depth. The linear actuator and soil sensor will be located at the backside of the robot. Due to this, the robot will rotate itself such that the linear actuator is facing the plant before lowering the sensor into the ground.

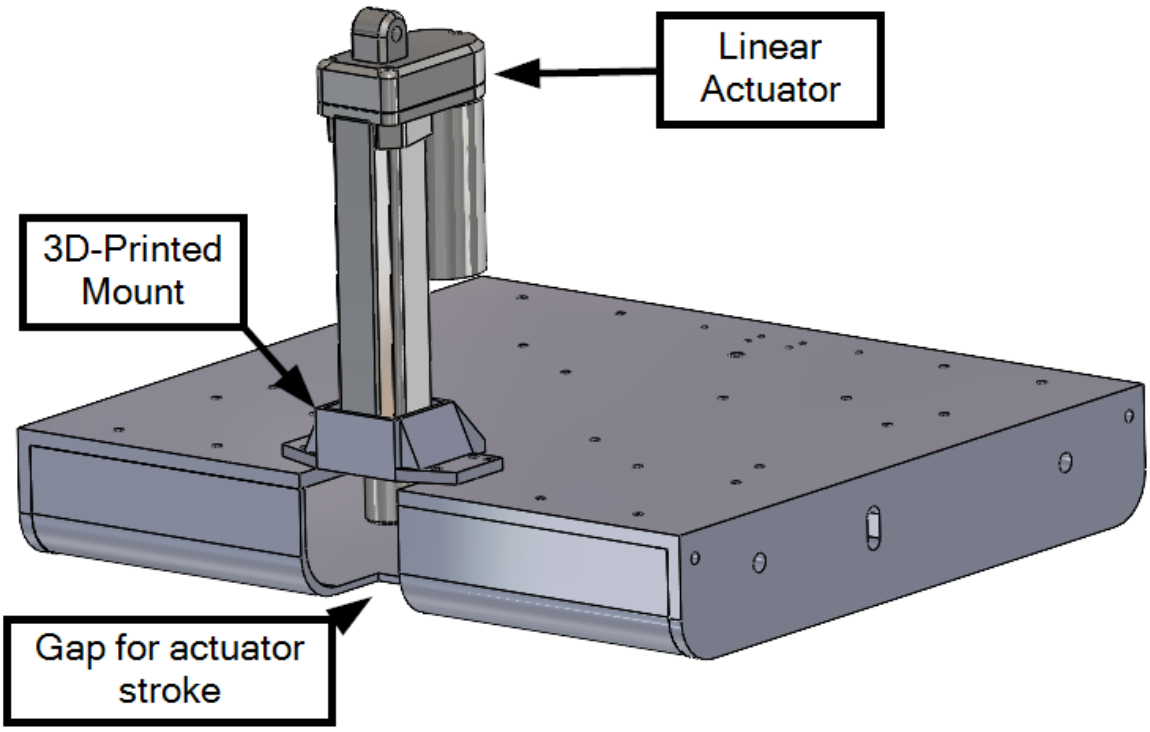


Figure 5: *View of linear actuator mounted on robot.*

Navigation Subsystem:

The ability of the robot to keep track of its current location and surroundings is critical for it to be fully autonomous. As such, we decided to research and quantify each localization approach to choose the best one for our robot.

Table 1: Localization Technique Trade Study

	Weights						
	0.25	0.15	0.2	0.15	0.05	0.2	
Localization Methods	Accuracy	Computational Cost	Reliability	Complexity	Implementation Costs	Adaptability	Final Score
GPS	8	5	4	3	1	6	5.25
Beacon-Based Localization Systems	7	6	6	4	7	4	5.6
Visual - Based SLAM	6	7	5	7	4	8	6.4

LiDAR SLAM	7	6	6	5	5	7	6.25
vSLAM and LiDAR SLAM	9	8	8	9	7	9	8.55

Table 1: *A table showcasing our localization trade study.*

From the trade study conducted above, we found that a combination of Visual Based SLAM and LiDAR SLAM would be appropriate for our project moving forward. We can leverage a lot of libraries and open source software such as ROS SLAM and OpenSLAM to be able to create our localization module. We will be mostly focusing on using the ROS SLAM toolbox to create, configure, and run our SLAM algorithms. With ROS, we would be able to leverage the ROS framework to enable simultaneous processing of the camera-based visual information and LiDAR-based depth data to ensure the robot's responsive time.

Since we will be using a combination of Visual SLAM and Lidar SLAM, we will have to use both a camera and a lidar sensor. The Stereo Camera from XYGStudy is capable of seeing in color, while sensing depth which provides us with more data to get accurate readings. We will also be incorporating a LiDAR tower in the middle of the robot, built from a 6" standoff and a 3D printed platform which the LiDAR is mounted to. We have put it above the rest of the components on the robot to avoid the robot interfering with the LiDAR's vision.

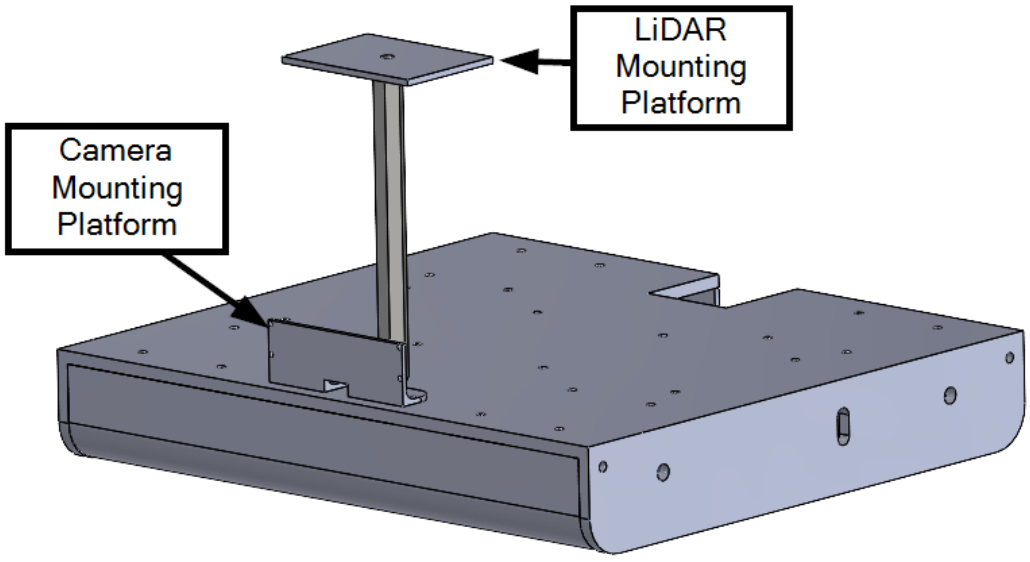


Figure 6: *LiDAR and Stereo Camera Mounts at the front of the chassis*

Chassis Design:

The chassis is built from two pieces of sheet metal, one a quarter inch thick and the other an eight inch thick. The quarter inch sheet is 3350mm x 3050mm. Two opposing sides of the sheet are bent 90 deg with a 20mm radius down to provide mounts for bearings on the motor axles. The motors are mounted to the

bottom of the quarter inch plate via Tetrix Motor Mounts. In the rear of the chassis, there is a 50mm x 50mm cutout for the linear actuator to reach through. The mounting holes on the chassis are pre-cut for mounting the intended hardware. There is a 3D printed mounting plate for the stereo camera in the front of the robot, and a 3D printed mount for the linear actuator located on the rear of the chassis. For both of these mounts, there are mounting holes integrated into the custom parts for the camera and the linear actuator to fasten into.

The thinner piece of sheet metal will be custom cut to also be bent 90 deg with a 20mm radius, but will also have mounting tabs with inserts for bolts to fasten into. This is meant to be a skid plate, so as to not expose the bottom of the robot to the elements. Furthermore, since the bottom of the robot is now protected from the soil, this provides sufficient space for the electronics to be mounted. As a result, the electronics (Jetson, motor controller, etc.) will be mounted to the bottom side of the quarter inch plate.

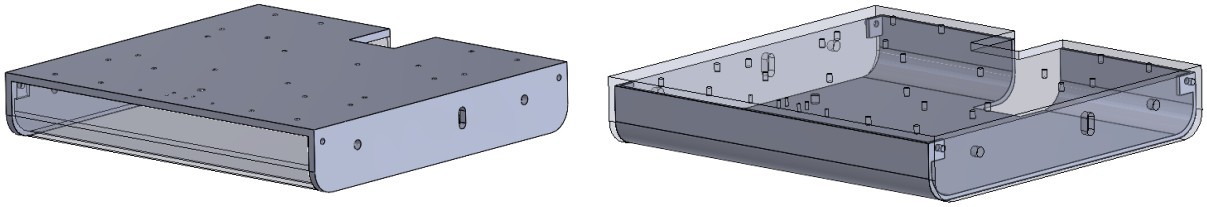


Figure 7: *Left, the top plate forming the chassis for the robot. Right, the bottom plate forming the skid plate and protection for the robot.*

Parts and Components

Part Number	Component Type	Model	Quantity	Unit Price
1	Battery	10,000 mAH Battery	1	90
2	Linear Actuator	Firgelli Automations 3" Stroke (35lbs)	1	119.99
3	Driver Motors	Tetrix-Max-TorqueNADO	4	29.95
4	Lidar	Slamtec 360 degree	1	99
5	Nozzle	McMaster 60 deg Cone	4	14.37
6	Drive Train	Tetrix Tread Kit	1	119.95
7	Stereo Camera	XYGStudy IMX219	1	56.99
8	Soil Sensor	Renke 4 in 1 Nutrient Sensor	1	50.9
9	Soil Sensor	Taidacent RS485	1	104

10	Safety	Emergency Stop	1	10
11	Computing Platform	NVIDIA Jetson Nano	1	149
12	Computing Platform	REV Robotics Control Hub	1	350
Total				1327.11

Fabrication and Assembly

Fault Recovery

The fault recovery mechanisms are designed to ensure robust performance and minimize the impact of unexpected issues. The robot is equipped with sensors, actuators, and intelligent algorithms to detect and recover from faults promptly. In addition, regular logs are generated and sent to the user which describe fault occurrences, system performance, and other information useful in preventing failures. In the case of very severe faults within the robot, manual control will be activated, allowing the user to stop the robot on their command and only continue with the processes on their command once the errors have been resolved.

Sensor Abnormalities

The robot relies on a soil moisture sensor to ensure optimal plant care. In the event of the soil moisture sensor reporting inconsistent data or erratic readings from the sensor, the robot activates an immediate response to prevent potential adverse effects on plants. The sensor will be defined as failing if the readings it gets from the soil sensor is outside of a predetermined range (found through basic experimental testing), or if the values have a sharp increase/decrease by more than 10% of the previously recorded values.

In the recovery plan, firstly, the robot temporarily halts the irrigation process to avoid overwatering or underwatering plants based on unreliable soil moisture readings. It does the same for nutrients, so as to not harm the plants by providing too much or too little nutrients. Afterwards, it triggers an alert notification through the software, indicating that the soil moisture sensor has failed. The notification includes details about the inconsistency in the readings and advises the user to inspect the sensor or take manual control if necessary. If the sensor is fixed, but faults continue to occur, the sensor will be recalibrated for the specific garden and/or the failure range will be modified.

Actuator Faults

The robot incorporates various actuators for essential functions such as movement and sensor deployment. Detection of actuator failures is critical to maintaining operational integrity. If the robot is outside of the garden path or moving below a minimal speed, irregularities in the actuators will be analyzed to determine if they are failing. Firstly, a safe state will begin, meaning that the robot temporarily halts any operation. For instance, if a wheel motor is erratic and causes the robot to turn excessively, the robot stops movement to prevent further damage to itself and to the garden, because safety is a top priority. Following this, the robot will activate a visual indicator such as an LED to indicate failure mode, as well as triggering an alert notification to the user.

Once the previous steps are completed, the robot will wait for human intervention because fixing actuator faults autonomously is larger than the scope of this project. For the user to fix these faults, each aspect of the actuators will be isolated and diagnostics will be run to determine the location of failure. These include:
- Analyzing feedback from motor encoders to determine rotational position and speed. Anomalies in encoder readings will lead to further investigation into the specific motor selection which may lead to a change in motors in the robot
- Checking electrical stress on the actuators by monitoring current draw and resistance. If excessive stress is detected, the power input will be reduced to prevent further damage.
- Checking mechanical stress on the actuators by looking for scrapes, cracks, etc. either through the robot itself or by obstacles in the environment causing damage to the robot. This will be fixed by cleaning up the environment of rocks or other debris or shifting the weight distribution of the robot away from the actuators that are more damaged.

Communication Faults
The robot relies on seamless communication for coordination and control. Detection of communication errors is paramount to maintaining connectivity. While in use, the robot continuously monitors the strength of the signals it receives. A significant drop in signal strength, complete loss of signals, packet errors, or other disruptions indicates a communication failure. The first step to handle this error is by having a communication retry protocol. After a loss of signal, the robot will pause any operations for ten seconds (for safety measures) and attempt to re-establish the connection. If the connection works again, the robot will continue its work through the garden. Although, if the connection is persistently spotty, it will have to be analyzed later by the user. If the connection fails to reconnect, an offline mode will be activated. While in this mode, the robot relies on pre-programmed routes throughout the garden and locally stored data to continue its essential functions autonomously. This is done as a very strong safety precaution and parts of this protocol are not entirely needed, but due to having a strong safety priority, this will be done to ensure no damage caused by a lack of connection.

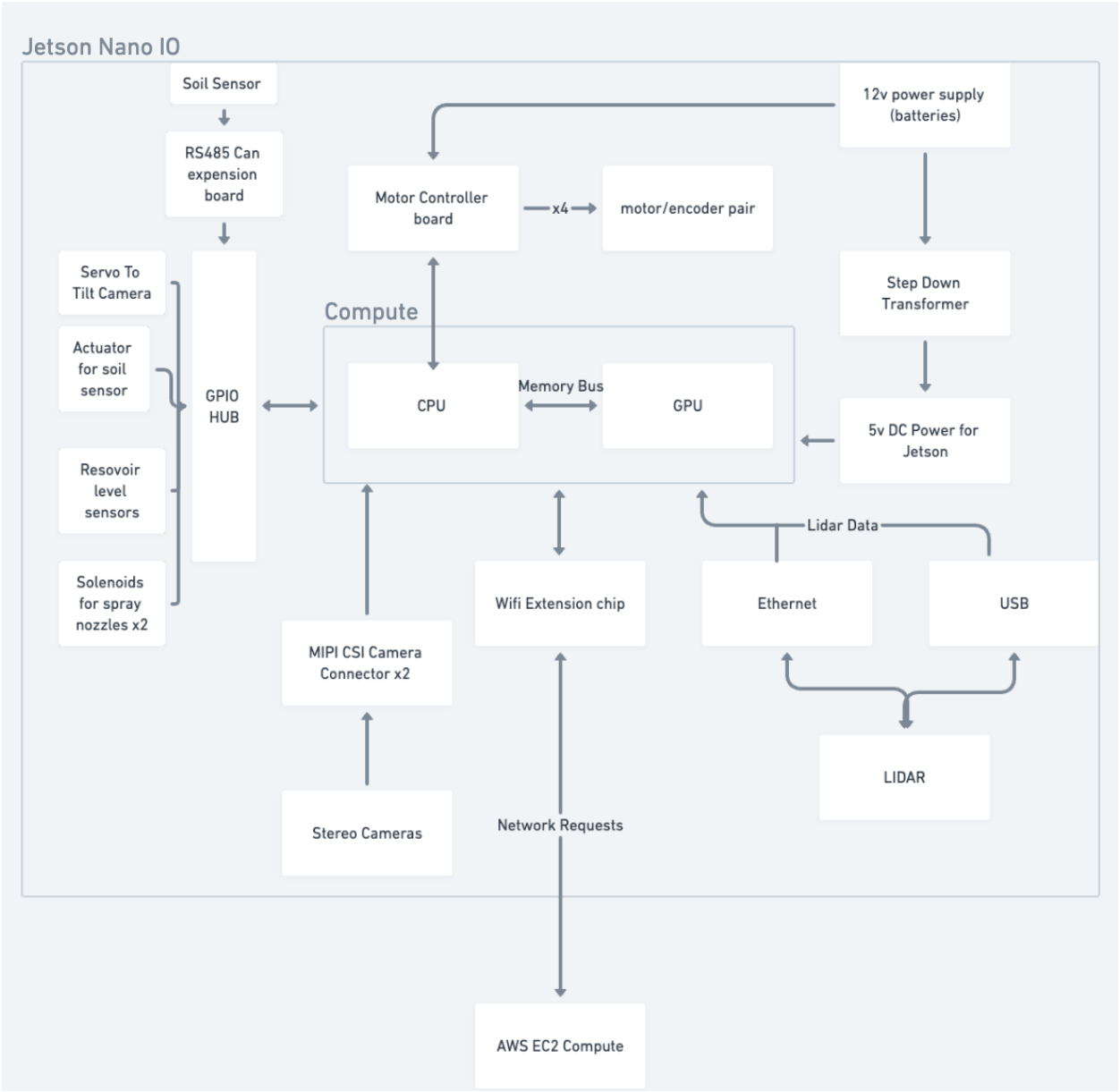
Degraded Operational Modes
For continuous and autonomous operation, the robot will transition into degraded modes. These are outlined below:

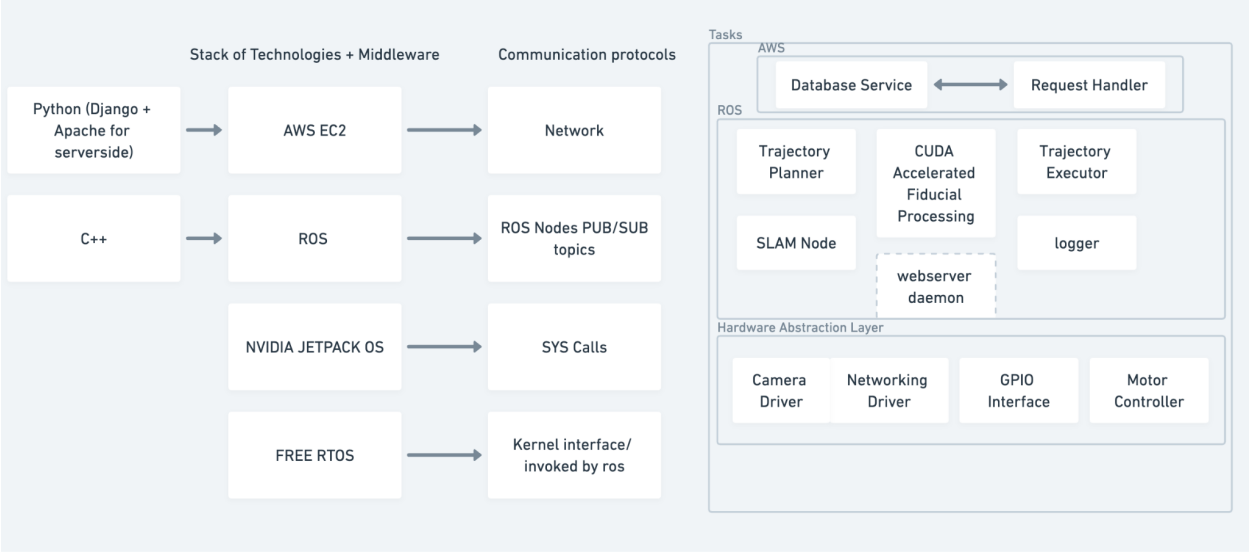
- ***Safe Navigation Mode:*** When there are sensor or navigation failures, the robot will reduce its speed (either slightly or completely) and widen the safety margin on the garden path to ensure full obstacle avoidance.
    - Activated by the described faults in the above section
    - Can be minimal or extreme. For example, in the actuator failure, the robot will be in extreme safe navigation mode, meaning no movement at all. However, for specific circumstances where the sensor is failing, the robot is just in minimal safe navigation mode, leading to a reduced speed and less functionality in the sensors, but still some operation. This is all done to optimize safety for the robot, user, and garden plants.
- ***Communication Failure Mode:*** As stated in the fault recovery section, the robot may have communication issues, and if so, it will enter the communication failure mode. As stated above,

the robot will autonomously execute predefined tasks or movements based on its last received actions, ensuring continued operation.

- ***Manual Control Mode:*** In critical situations (mostly scenarios with very low probabilities of occurring but still possible), the robot will shut off all autonomous behavior and resort to full manual mode. This means that the user has full control over the robot's functionality and only the user is able to restart the robot in autonomous mode.
- ***Lower Power Mode:*** When the robot measures its power to be low, it will conserve energy and prioritize essential functions. For instance, the robot may turn off any smart navigation and traverse along predefined paths to save algorithmic time and energy. Alternatively, the robot may pause the nutrient dispensal and only dispense water to the plants. Finally, the robot can lower its speed to conserve energy until it is recharged.
- ***Diagnostic Mode:*** When a failure mode is detected and user intervention is required to fix the issue, the robot will enter the diagnostic mode. The robot will travel to the outside edge of the garden and turn off. This enables the user to be able to repair the robot without causing damage to the garden or causing any other difficulties. Once the user fixes any issues, it will restart the robot and put it back in the garden in its standard autonomous mode to continue its operations. This mode will also be used when the robot needs to be refilled on either water or nutrients, allowing the user to replenish the robot and then have it continue performing its actions.

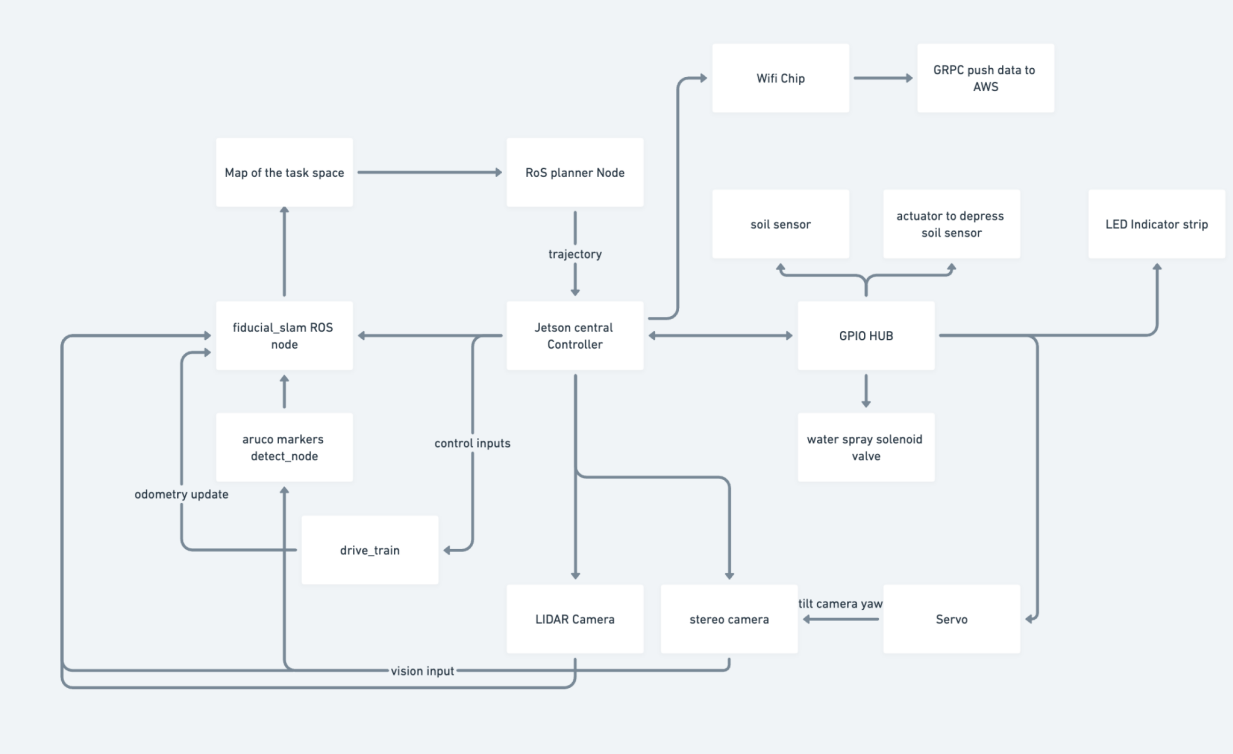
Robot Software Architecture


Jetson Nano IO
Soil Sensor
RS485 Can expension board
12v power supply (batteries)
Motor Controller board
x4
motor/encoder pair
Servo To Tilt Camera
Actuator for soil sensor
Resovoir level sensors
Solenoids for spray nozzles x2
GPIO HUB
Compute
CPU
Memory Bus
GPU
Step Down Transformer
5v DC Power for Jetson
Lidar Data
Wifi Extension chip
Ethernet
USB
MIPI CSI Camera Connector x2
LIDAR
Stereo Cameras
Network Requests
AWS EC2 Compute



This diagram outlines the Stack of technologies and the communication protocols to communicate with them along with separating these persistent running tasks into their own nodes and placing them at their appropriate layer. By having this hardware abstraction layer we can design our code+ros environment to be agnostic to the actual hardware specific implementations and drivers of our robot components.

Lastly the Below diagram Outlines the actual operation of the jetson compute and the handshake between data/io and various tasks that will be running



From these above diagrams key software components have been isolated and pseudocode has been generated:

SLAM system based on ubiquity robotics fiducial slam we will derive our own slam class that inherits from their fiducial slam class:
https://github.com/UbiquityRobotics/fiducials/tree/kinetic-devel/fiducial_slam

```
class FeatureExtractor:
    def extract_features(self, image):
        """
        Extracts features from the provided image.
        :param image: Image from robot's camera
        :return: List of detected features
        """
        # Implement feature extraction logic (e.g., corners, edges)
        features = []
        # ... feature extraction logic ...
        return features

class DataAssociation:
    def associate(self, features, previous_features):
        """
        Associates current detected features with previous features.
        :param features: Current frame features
        :param previous_features: Previous frame features
        :return: List of associated feature pairs
        """
        # Implement data association logic
        associations = []
        # ... data association logic ...
        return associations

class PoseEstimator:
    def estimate_pose(self, associations):
        """
        Estimates the current pose of the robot based on feature
associations.
        :param associations: Associations between current and previous
features
        :return: Estimated pose (x, y, theta)
        """
        # Implement pose estimation logic
        pose = (0, 0, 0)  # Placeholder for the estimated pose
```

```
        # ... pose estimation logic ...
        return pose

class MapUpdater:
    def update_map(self, pose, features):
        """
        Updates the map based on the estimated pose and current features.
        :param pose: Estimated robot pose
        :param features: Current frame features
        :return: Updated map
        """
        # Implement map updating logic
        updated_map = None
        # ... map updating logic ...
        return updated_map

class SLAMSystem:
    def __init__(self):
        self.feature_extractor = FeatureExtractor()
        self.data_association = DataAssociation()
        self.pose_estimator = PoseEstimator()
        self.map_updater = MapUpdater()
        self.previous_features = None
        self.map = None
        self.pose = (0, 0, 0)  # Initial pose (x, y, theta)

    def process_frame(self, image):
        """
        Processes an incoming frame from the robot's camera and updates the
SLAM state.
        :param image: Image from robot's camera
        """
        features = self.feature_extractor.extract_features(image)
        if self.previous_features is not None:
            associations = self.data_association.associate(features,
self.previous_features)
            self.pose = self.pose_estimator.estimate_pose(associations)
            self.map = self.map_updater.update_map(self.pose, features)
        self.previous_features = features

# Example usage:
slam_system = SLAMSystem()
while True:
```

```
    image = get_camera_image()
    slam_system.process_frame(image)
    # Now slam_system.pose and slam_system.map hold the latest pose and map
```

The full code would be written in C++ for the performance demands of SLAM but we are using Python to illustrate the OOP structure in a more readable way. The above class structure instantiates the key components of our slam systems and prototypes the docstrings and expected returns we will be exchanging in the slam control loop that we highlighted in our concept design. This slam loop runs 'every frame' which will be clocked by the LIDAR sensors frequency given that it's the slowest component of the slam system. The frame process function handles the control loop of estimating the robot's position as well as updating the map.

Fiducial Marker detection. We will be using ROS' built in aruco fiducial marker detection so that we don't need to reinvent the wheel. The general schema of what the fiducial marker detector would look like as its ross node would look like: where detected markers would be published under their associated topic that can be accessed by a subscriber in the SLAM ROS node.

```
class FiducialMarkerDetector:
    def __init__(self):
        self.detector = create_detector_instance()  # Assuming a library
function

    def detect_markers(self, image):
        # Detect markers in the given image
        markers = self.detector.detect(image)
        return markers


# Example usage
detector = FiducialMarkerDetector()
image = get_camera_image()
markers = detector.detect_markers(image)
publish(detector_topic,markers)
```

Soil Sensor:

```
class SoilAnalysisModule:
    def __init__(self, soil_sensor):
        self.soil_sensor = soil_sensor

    def analyze_soil(self):
        # Get soil sensor reading
```

```
        soil_data = self.soil_sensor.read()
        # Analyze data
        analysis = self._interpret_data(soil_data)
        return analysis

    def _interpret_data(self, soil_data):
        # Logic to interpret soil data
        pass

# Example usage
soil_sensor = SoilSensor()
soil_analysis_module = SoilAnalysisModule(soil_sensor)
soil_health = soil_analysis_module.analyze_soil()
```

The sensor would be a lower level GPIO module wrapper and we'd have a higher level class that would call the method of the soil sensor to make it hardware independent. Then with this sensor data we can read, interpret and publish the data.

Watering System:

```
class WateringSystemController:
    def __init__(self, valve_controller):
        self.valve_controller = valve_controller

    def water_plants(self, soil_moisture_level):
        if soil_moisture_level < MOISTURE_THRESHOLD:
            self.valve_controller.open_valve()
        else:
            self.valve_controller.close_valve()

# Example usage
valve_controller = ValveController()
watering_system = WateringSystemController(valve_controller)
watering_system.water_plants(soil_health['moisture_level'])
```

The valve controller will be an api wrapper to handle the solenoid valves for the nozzles to keep things hardware independent, from a soil moisture level that can be accessed by this node through subscribing to the soil sensor's topic and using that information to calibrate how much water it wants to dispense, using the current moisture level as a conditional check

Website system:

```
# soil_monitor/views.py
```

```python
from django.shortcuts import render
from django.http import JsonResponse
from .models import SoilData

def soil_data(request):
    # Assuming we have a model called SoilData with soil information
    data = SoilData.objects.all().values()
    return JsonResponse(list(data), safe=False)

def soil_data_view(request):
    # Render a template to display soil data
    return render(request, 'soil_monitor/soil_data.html')
```

We will be leveraging Django in python for its OOP extensibility to develop the web server to host the user's garden information

Django uses the MVC paradigm. model, view, controller. The model interacts with the database, the view is the ui served to the user and the controller selects the appropriate view based on the user's request and the contingent data in the database. These html files operate with the django template language so we can insert user data.

Url mapping:

```python
# garden_monitor/urls.py
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('soil/', include('soil_monitor.urls')),
    path('images/', include('garden_images.urls')),
]
```

Django model - database schemas. Django allows you to use python classes to define the schemas for database entries. Below are the key schemas the end website will use.

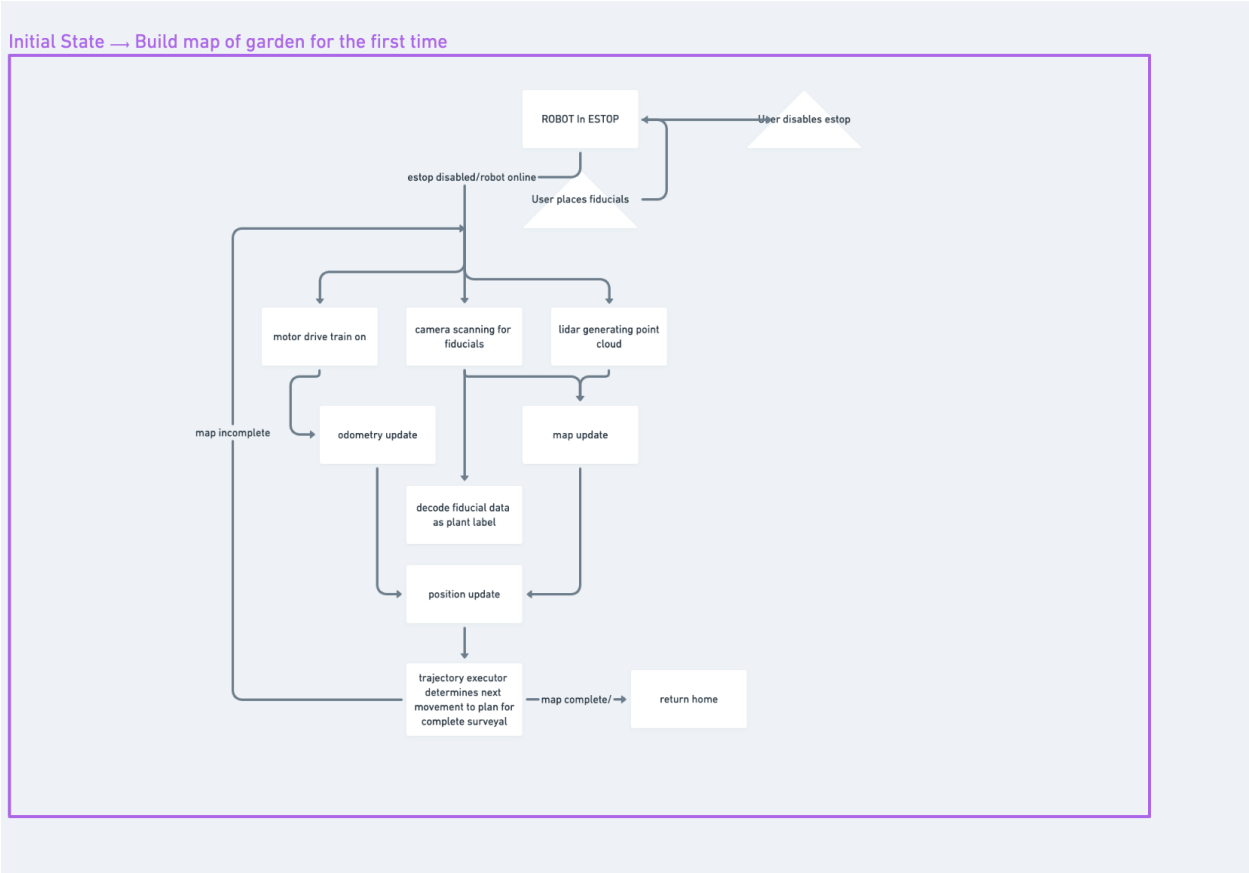
Garden plot maps the user to their unique garden plot, Soil data contains the timestamp, ph level temperature and moisture content, the timestamp field will allow for graphing and filtering by time period. The garden plot image encodes the image of each plant plot in the garden timestamped to allow for the user to animate these images of a plant over time as a form of timelapse.

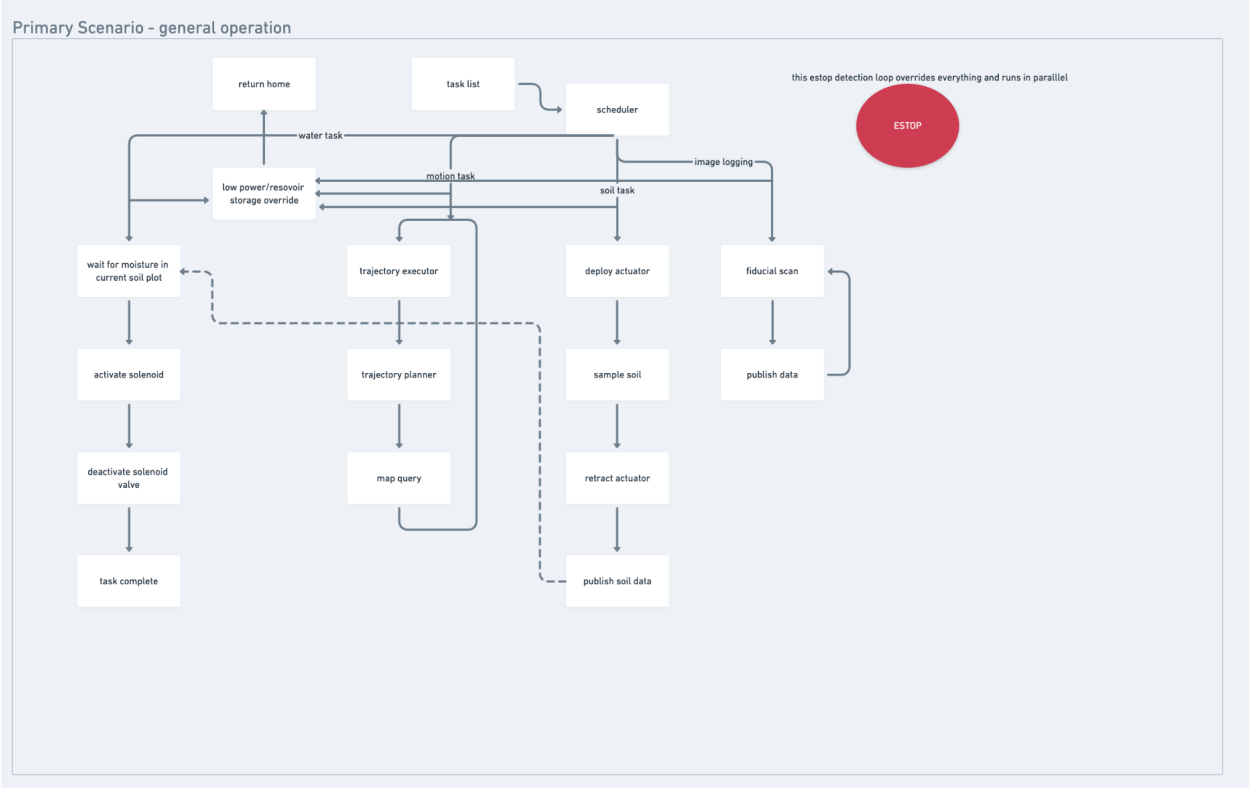
```python
from django.db import models
from django.contrib.auth.models import User

class GardenPlot(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    name = models.CharField(max_length=100)
    description = models.TextField(blank=True)

    def __str__(self):
        return self.name

class SoilData(models.Model):
    garden_plot = models.ForeignKey(GardenPlot, related_name='soil_data',
on_delete=models.CASCADE)
    timestamp = models.DateTimeField(auto_now_add=True)
    ph_level = models.DecimalField(max_digits=5, decimal_places=2)
    temperature = models.DecimalField(max_digits=5, decimal_places=2)
    moisture_content = models.DecimalField(max_digits=5, decimal_places=2)

    def __str__(self):
        return f"{self.garden_plot.name} data at {self.timestamp}"

class GardenPlotImage(models.Model):
    garden_plot = models.ForeignKey(GardenPlot, related_name='images',
on_delete=models.CASCADE)
    timestamp = models.DateTimeField(auto_now_add=True)
    image = models.ImageField(upload_to='garden_plots/')

    def __str__(self):
        return f"{self.garden_plot.name} image at {self.timestamp}"
```

Temporal State:

Below outlines our approach to managing temporal state and concurrency in more detail than the application level diagram from the software architecture

This statechart captures the initial scenario of the robot being initialized by the user and building its map of the garden for the first time. The triangle denotes user behavior and the rectangles are the robot state and actions we can see the concurrency in the fanout of the localisation tasks.





This is a state chart that captures the normal operation we have these two types of tasks that are persistent e.g motion tasks and fiducial resolution as well as event driven tasks such as soil sensing and watering which need to be schedule and do not run persistently in the same way. However, they can run consistently with persistent tasks as shown in the above diagram. The main case of blocking where one process is waiting for the other is the water task which requires a reading on the soil moisture content to actually determine how much water to dispense. The task list is what is generated by the general application of the robot which handles parsing user requested watering schedules and frequency and converting these into task descriptions for the robot through its api. The scheduler deploys and interleaves these concurrent tasks in parallel enabling the robot to successfully execute on its functionality.

Safety

Emergency Stop (Estop):

- A hardware emergency stop button on the robot. Ensure that it's easily accessible, well-lit, and marked clearly for quick identification and action in case of an emergency. This is our fail-safe mechanism so it is important that the user is very clear on where it is to start.--> to make our design more human centric and the existence of hte estop more prominent we will use a red push/twist estop button where the robot is started and turned on via twisting the estop button to release it and then engage it by pushing it down so the user must interact with the ESTOP prior to operating the system

Battery Safety and Precautions -

- We will advertise a clear temperature disclaimer on the operating conditions of the battery and add a moisture sensor/indicator paper as a safety net to detect water entering the battery housing

Remote stop:

- Outline to the user how they can remotely start and stop the robot

Staying out of the robot's workspace:

- Safety disclaimer on avoiding the robot when it is actively spraying water