Farms Manager - Distributed Systems for IoT

Version 1.0

Table of Contents

Welcome to Farms Manager's Documentation

Welcome to the documentation of the Farms Manager project. This document provides a comprehensive guide to understanding, using, and developing a smart management system for unmanned autonomous vehicles in precision agriculture on edge.

In this documentation, you will find detailed explanations of the system's architecture, core functionalities, and deployment processes. Additionally, the document covers the usage instructions to facilitate smooth integration and further development.

Whether you are a developer, researcher, or end-user, this documentation aims to serve as a valuable resource for understanding and extending the capabilities of the Farms Manager system.

What is Farms Manager?

Farms Manager is an innovative project designed to create a smart management system for unmanned autonomous vehicles used in precision agriculture. The system leverages edge computing to enhance the efficiency and effectiveness of agricultural operations, enabling real-time data processing and decision-making directly on the field.

In each farm, there is a central node that coordinates multiple autonomous vehicles (Fleet Manager), such as drones and ground robots. These vehicles are equipped with various sensors and tools to perform tasks like monitoring crop health, soil analysis, and automated planting or harvesting.

The fleet manager comunicates with the vehicles via ROS2 actions, collects data, and executes commands to optimize farming operations. All the fleet managers are connected to Node Red via MQTT.

Node Red provides a user interface for farmers and agricultural experts. This interface allows users to monitor the status of their missions and its results.

Missions and Commands

A mission is a high-level task assigned to a drone of a specific farm, which consists of a series of commands that the autonomous vehicle must execute. The commands are the specific actions that the vehicle performs to accomplish the mission.

Examples of commands include:

- Takeoff: The vehicle takes off from the ground.
- Go to Waypoint: The vehicle flies to a specified GPS coordinate.
- Capture Image: The vehicle captures an image of the crops.
- Analyse Image: The vehicle processes the captured image.
- Spray Treatment: The vehicle sprays the specific treatment on the crops.
- Return Home: The vehicle returns to its starting point.
- Land: The vehicle lands safely on the ground.

Getting Started

How to run

Requirements

- Laptop/PC with Docker installed (Follow the Docker Installation tutorial to install Docker on your Windows machine).
- Laptop/PC with Node-Red installed (Follow the Node-RED Installation tutorial to install Node-Red on your Windows machine).
- Raspberry Pi 5 with Raspberry Pi OS (64-bit). (Optional)

Fleet Manager

If it is going to run on the Laptop/PC with Windows 11, start `Docker Desktop` .

In the `Fleet Manager` directory, run the following command:

```
docker network create --driver=bridge --subnet=192.168.100.0/24 --
gateway=192.168.100.1 fleet_net
```

Before executing the Fleet Manager, make sure to change the environment variables in the `.env` file to avoid conflicts if you are running multiple fleet managers.

```
FARM_ID=1
MQTT_BROKER_ADDR=10.138.137.150
MQTT_PORT_NUM=1885
```

To start the `Fleet Manager` , execute:

```
docker compose build
docker compose up
```

Drone

If it is going to run on the Laptop/PC with Windows 11, start `Docker Desktop` .

In the `Drone` directory, run the following command (If it's the same machine as the Fleet Manager, skip this step):

```
docker network create --driver=bridge --subnet=192.168.100.0/24 --
gateway=192.168.100.1 fleet_net
```

Before executing the drone, make sure to change the `DRONE_ID` environment variables in the `.env` file to avoid conflicts if you are running multiple drones.

```
DRONE_ID=1
```

To execute the `Drone` , execute:

```
docker compose build
docker compose up
```

Start Node-Red

To start Node-Red, open a terminal and execute:

```
node-red
```

You can open your browser and navigate to:

```
http://localhost:1880
```

Install the palette `node-red-dashboard` , `node-red-contrib-aedes` and `node-red-contrib-web-worldmap` in Node-Red.

Import the Node-Red json flow that is in the `Node-Red` directory and deploy it.

To see the Node-Red environment, open your browser and navigate to:

```
http://localhost:1880/dashboard
http://localhost:1880/ui
http://localhost:1880/worldmap/
```

Docker Installation

This guide covers the installation of Docker on two different platforms: Docker Desktop for Windows and Docker Engine for Raspberry Pi 5 (RPI5).

Docker Desktop for Windows

Download and Install

1. Visit the official Docker Desktop download page:

   https://www.docker.com/products/docker-desktop/

2. Download the **Docker Desktop for Windows** installer.

3. Run the installer and follow the on-screen instructions.

4. After installation, start Docker Desktop from the Start menu.

Verify Installation

To verify Docker is working, open **PowerShell** and run:

```
docker run hello-world
```

You should see a message confirming that Docker is installed and running correctly.

Docker Engine for RPI5

Set up Docker's APT repository

Add Docker's official GPG key:

```
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/raspbian/gpg -o /
etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
```

Add Docker's Repository to APT Sources:

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/
keyrings/docker.asc] https://download.docker.com/linux/raspbian \
  $(. /etc/os-release && echo \"$VERSION_CODENAME\") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

Reboot the RPI5.

Install the Docker packages

To install the latest versions of Docker components, run:

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-
buildx-plugin docker-compose-plugin
```

Verify the installation

To confirm Docker is installed and working correctly, run:

```
sudo docker run hello-world
```

You should see a message confirming successful installation.

Add Docker sudo privileges

To manage Docker as a non-root user, add your user to the *docker* group:

```
sudo usermod -aG docker $USER
```

Log out and log back in for the changes to take effect.

References

- Docker Desktop for Windows documentation: https://docs.docker.com/desktop/install/windows-install/
- Docker Documentation – Install Docker Engine on Raspberry Pi OS: https://docs.docker.com/engine/install/raspberry-pi-os/

Node-RED Installation

This guide explains how to install **Node.js** and **Node-RED** on Windows using **Chocolatey** and **npm** .

Install Node.js

Download and Install

1. Open **bash** as Administrator.

2. Install **Chocolatey** (a Windows package manager):

```
bash -c "irm https://community.chocolatey.org/install.ps1|iex"
```

3. Use Chocolatey to install **Node.js** version 24.11.1:

```
choco install nodejs --version="24.11.1"
```

4. Verify the installation:

```
node -v    # Should print "v24.11.1"
npm -v     # Should print "11.6.2"
```

Reboot the laptop after installation.

Install Node-RED

After Node.js is installed, use **npm** to install Node-RED globally:

```
npm install -g --unsafe-perm node-red
```

Reboot the laptop once the installation completes.

Run Node-RED

To start Node-RED, open **bash** and run:

```
node-red
```

This will start the Node-RED server. You can then open your browser and navigate to:

```
http://localhost:1880
```

References

- Node-RED Official Documentation (On Windows): https://nodered.org/docs/getting-started/windows
- Node.js Official Website (Download): https://nodejs.org/en/download

Code documentation

Fleet Manager

Init Fleet Manager

This module initializes and runs the Fleet Manager system for a specific farm.

It performs the following main tasks: 1. Creates a thread-safe logger for recording system activity. 2. Initializes a ROS2 Action Client used for communicating with drones. 3. Initializes the MQTT publisher client. 4. Starts the MQTT subscriber to receive mission commands from Node-RED (or other sources). 5. Keeps the Fleet Manager running continuously until manually terminated.

This module serves as the entry point for the Fleet Manager container in the distributed UAV control architecture, coordinating MQTT and ROS2 communication layers.

init_fleet. create_logger (*logger_name : str* , *log_file : str*) → Logger [source]

Creates and configures a thread-safe logger for the Fleet Manager system.

The logger writes logs both to the console and to a rotating log file. It uses *ConcurrentRotatingFileHandler* to ensure safe access from multiple threads.

Parameters :

- **logger_name** (*str*) – The name to assign to the logger instance.
- **log_file** (*str*) – Path to the log file where entries will be stored.

Returns :

A configured logger instance.

Return type :

logging.Logger

init_fleet. main (*farm_id : int* , *mqtt_broker : str* , *mqtt_port : int*) → None [source]

Initializes and runs the Fleet Manager system for a specific farm.

This function sets up the logger, initializes the ROS2 action client, and starts the MQTT subscriber. Once initialized, it keeps the system running until the user stops it manually (e.g., with Ctrl+C).

Parameters :

- **farm_id** (*int*) – Unique identifier of the farm instance.
- **mqtt_broker** (*str*) – Hostname or IP address of the MQTT broker.
- **mqtt_port** (*int*) – Port number of the MQTT broker.

Returns :

None

Return type :

None

Commands Action Client

This module manages the communication between the Fleet Manager and the drones using ROS2 Actions. It supports automatic drone discovery, concurrent communication with multiple drones, and integrates with MQTT for mission status reporting.

Main responsibilities: 1. Automatically discovers and connects to drones via ROS2 action topics. 2. Sends mission goals (command sequences) to specific drones. 3. Handles feedback and results asynchronously using callbacks. 4. Reports mission progress and outcomes to the Fleet Manager via MQTT.

It serves as the core bridge between the ROS2-based drone layer and the MQTT-based Fleet Manager system.

class commands_action_client. CommandsActionClient (** args : Any* , *** kwargs : Any*) [source]

Bases: `Node`

ROS2 Action Client node for managing communication with multiple drones.

This class enables the Fleet Manager to send missions to drones, receive feedback, and handle mission results. It dynamically creates and manages multiple ROS2 Action Clients (one per connected drone).

Parameters :

- **farm_id** (*int*) – Unique identifier for the farm instance associated with this Fleet Manager.
- **fleet_logger** (*logging.Logger*) – Logger instance used for recording Fleet Manager events.

__init__ (*farm_id* , *fleet_logger*) [source]

action_clients

Dictionary mapping drone IDs to their respective ActionClient instances.

drone_busy

Dictionary tracking whether each drone is currently executing a mission.

farm_id

Farm ID corresponding to this Fleet Manager instance.

fleet_logger

Logger for system activity.

ros2_action_queue

Queue used for internal ROS2 action event management.

send_goal (*drone_id* , *commands_list*) [source]

Sends a mission (list of commands) to the specified drone via ROS2.

If the drone is already executing another mission, the request is rejected. Otherwise, a new ROS2 Action Client is created if necessary, and the mission is transmitted asynchronously.

Parameters :

- **drone_id** (*int*) – Identifier of the target drone.
- **commands_list** (*list [str]*) – List of mission commands to be executed by the drone.

Returns :

None

Return type :

None

goal_response_callback (*future* , *drone_id*) [source]

Callback executed when the drone responds to a mission goal request.

Handles both acceptance and rejection of the mission by the drone. If accepted, it attaches a callback to handle mission result retrieval.

Parameters :

- **future** (*rclpy.task.Future*) – Future object representing the asynchronous goal response.
- **drone_id** (*int*) – Identifier of the drone that responded to the mission.

Returns :

None

Return type :

None

get_result_callback (*future* , *drone_id*) [source]

Callback executed when the drone finishes executing the mission.

Publishes the mission result to MQTT and marks the drone as available again.

Parameters :

- **future** (*rclpy.task.Future*) – Future object containing the mission result.
- **drone_id** (*int*) – Identifier of the drone that completed the mission.

Returns :

None

Return type :

None

feedback_callback (*feedback_msg* , *drone_id*) [source]

Callback executed when the drone provides feedback during mission execution.

Each feedback message is logged and published to the MQTT topic corresponding to the farm.

Parameters :

- **feedback_msg** (*commands_action_interface.action.Commands.FeedbackMessage*) – ROS2 feedback message from the drone.
- **drone_id** (*int*) – Identifier of the drone sending the feedback.

Returns :

None

Return type :

None

commands_action_client. create_ros2_action_client (*farm_id* , *fleet_logger*) [source]

Initializes the ROS2 environment and creates a multi-threaded Action Client for managing drone communication.

The Action Client runs in a separate thread, allowing concurrent handling of ROS2 callbacks without blocking the main Fleet Manager process.

Parameters :

- **farm_id** (*int*) – Unique identifier for the farm instance.
- **fleet_logger** (*logging.Logger*) – Logger instance used for recording Fleet Manager events.

Returns :

Instance of `CommandsActionClient` ready to communicate with drones.

Return type :

CommandsActionClient

commands_action_client. parse_feedback_data (*data*) [source]

Attempts to convert the incoming data to a dict. Accepts:

- dict objects directly
- JSON strings
- strings using single quotes like a Python dict
- lists of characters (as you're receiving)

MQTT Components

MQTT Subscriber

This module manages the MQTT subscription component of the Fleet Manager system. It listens for incoming mission commands from Node-RED (or test tools) and forwards them to the appropriate drones via the ROS2 Action Client.

Main responsibilities: 1. **Subscribe** to farm-specific MQTT topics (e.g., *farm/<farm_id>*) with QoS 2. 2. **Receive** mission messages in JSON format. 3. **Parse** mission data to extract *vehicleId* and the list of commands. 4. **Forward** parsed commands to the ROS2 Action Client for drone execution. 5. **Set Last Will** for unexpected disconnections.

Expected message format:

The MQTT subscriber expects payloads with the following JSON structure:

```
{
    "vehicleId": 1,
    "commands": [
        {
            "missionId": 101,
            "commandId": 1,
            "commandType": "TAKEOFF",
            "params": {"altitude": 10}
        },
        {
            "missionId": 101,
            "commandId": 2,
            "commandType": "MOVE",
            "params": {"x": 20, "y": 30}
        }
    ]
}
```

Once received, the subscriber automatically converts each command into a stringified JSON element and passes them to the ROS2 layer for dispatch to the corresponding drone.

mqtt_sub. start_mqtt_sub (*farm_id* , *ros2_client* , *fleet_logger* , *mqtt_broker : str* , *mqtt_port : int*) [source]

Starts an MQTT subscriber for the specified farm.

This function connects to the MQTT broker, subscribes to the topic corresponding to the given farm ID with **Quality of Service 2 (QoS 2)** , and sets a **Last Will and Testament (LWT)** message for unexpected disconnections.

Parameters :

- **farm_id** (*int*) – Unique identifier of the farm instance.
- **ros2_client** (*CommandsActionClient*) – Instance of the ROS2 Action Client used to send goals to drones.
- **fleet_logger** (*logging.Logger*) – Logger instance used for recording system activity.
- **mqtt_broker** (*str*) – Hostname or IP address of the MQTT broker.
- **mqtt_port** (*int*) – Port number of the MQTT broker.

Returns :

Configured MQTT client instance.

Return type :

paho.mqtt.client.Client

MQTT Publisher

This module provides utility functions for publishing messages to the MQTT broker from the Fleet Manager system. It acts as the outbound communication interface, sending updates to Node-RED or other subscribers.

It handles four main types of messages: 1. **Goal messages** – Sent when a drone accepts a mission. 2. **Feedback messages** – Sent to report intermediate progress from the drone. 3. **Result messages** – Sent when a mission has been completed. 4. **Warning messages** – Sent to notify about system-level issues or unavailable drones.

All functions share the same structure: - Build a JSON payload with a textual message. - Publish it to a topic associated with a specific farm or drone. - Log the publication through the Fleet Manager's logger.

mqtt_pub. MQTT_KEEPALIVE = *60*

Keepalive interval in seconds for the MQTT connection.

mqtt_pub. mqtt_client = *None*

Global MQTT client instance used for publishing all messages. Initialized in initialize_publisher_client.

mqtt_pub. initialize_publisher_client (*farm_id* , *fleet_logger* , *mqtt_broker : str* , *mqtt_port : int*) → None [source]

Initializes the global MQTT publishing client, configures the Last Will and Testament (LWT), connects to the broker, and starts the network loop.

This function must be called once before any publishing operations.

Parameters :

- **fleet_logger** (*logging.Logger*) – Logger instance for recording system activity.
- **mqtt_broker** (*str*) – Hostname or IP address of the MQTT broker.
- **mqtt_port** (*int*) – Port number of the MQTT broker.

Returns :

None

Return type :

None

mqtt_pub. mqtt_goal (*farm_id* , *drone_id* , *message* , *fleet_logger*) [source]

Publishes a “goal” message to the MQTT broker with **QoS 2** .

This function is typically called when a drone accepts a mission request. The message is sent to the topic corresponding to the specific farm and drone.

Parameters :

- **farm_id** (*int*) – Unique identifier of the farm.
- **drone_id** (*int*) – Unique identifier of the drone.
- **message** (*str*) – Message content to publish (usually mission confirmation).
- **fleet_logger** (*logging.Logger*) – Logger instance for recording system activity.

Returns :

None

Return type :

None

mqtt_pub. mqtt_feedback (*farm_id* , *drone_id* , *message* , *fleet_logger*) [source]

Publishes a “feedback” message to the MQTT broker with **QoS 2** .

This function reports intermediate mission updates from the drone to the Fleet Manager. Feedback messages indicate command progress or partial mission states.

Parameters :

- **farm_id** (*int*) – Unique identifier of the farm.
- **drone_id** (*int*) – Unique identifier of the drone.
- **message** (*str*) – Message content to publish (usually feedback information).
- **fleet_logger** (*logging.Logger*) – Logger instance for recording system activity.

Returns :

None

Return type :

None

mqtt_pub. mqtt_result (*farm_id* , *drone_id* , *message* , *fleet_logger*) [source]

Publishes the final mission result to the MQTT broker with **QoS 2** .

This message indicates that a mission has been completed by the drone and includes the outcome or status of the plan execution.

Parameters :

- **farm_id** (*int*) – Unique identifier of the farm.
- **drone_id** (*int*) – Unique identifier of the drone.
- **message** (*str*) – Message describing the mission result.
- **fleet_logger** (*logging.Logger*) – Logger instance for recording system activity.

Returns :

None

Return type :

None

mqtt_pub. mqtt_warning (*farm_id* , *message* , *fleet_logger*) [source]

Publishes a warning message to the MQTT broker at the farm level with **QoS 2** .

Warnings are not associated with a specific drone. They are used to communicate system issues such as unavailable drones, communication errors, or initialization problems.

Parameters :

- **farm_id** (*int*) – Unique identifier of the farm.
- **message** (*str*) – Warning text to publish.

- **fleet_logger** (*logging.Logger*) – Logger instance for recording system activity.

Returns :

None

Return type :

None

Drone

Init Drone

This module initializes the ROS2 drone system and starts the drone's Action Server. It is the entry point for the **Drone Container** in the distributed UAV Fleet Management architecture.

Main responsibilities: 1. **Initialize logging** for each drone instance (both console and rotating file logs). 2. **Start the ROS2 Action Server** , enabling communication with the Fleet Manager. 3. **Graceful shutdown** when the process is stopped.

The drone instance listens for mission goals sent via ROS2 Actions and simulates their execution step-by-step, reporting feedback and results back to the Fleet Manager.

Usage example:

```
python init_drone.py --drone_id 1
```

This will initialize a drone node with ID 1, creating a dedicated log file in `Logs/Drone_1.log` and registering its ROS2 Action Server.

init_drone. create_logger (*logger_name : str* , *log_file : str*) → Logger [source]

Creates and configures a logger instance for the drone.

The logger outputs to both console and file (with rotation). Each drone maintains its own log file for easier debugging and traceability.

Parameters :

- **logger_name** (*str*) – Name of the logger instance (typically the drone ID).
- **log_file** (*str*) – Path to the log file.

Returns :

Configured logger instance.

Return type :

logging.Logger

init_drone. main (*drone_id : int*) → None [source]

Main entry point for the drone system.

Initializes the ROS2 Action Server for the given drone ID and begins listening for mission commands from the Fleet Manager.

Parameters :

drone_id (*int*) – Unique identifier of the drone instance.

Returns :

None

Commands Action Server

This module implements the **ROS2 Action Server** for each drone in the fleet. It defines how the drone receives mission commands from the Fleet Manager via ROS2 Actions, processes them sequentially, and reports real-time feedback and final results.

Main responsibilities:

1. **Initialize** a ROS2 Action Server in a drone-specific namespace (*/drone<ID>/commands*).
2. **Receive** missions as lists of commands from the Fleet Manager.
3. **Execute** each command using the *process_command* function.

4. **Publish feedback** for each executed command.

5. **Return results** once the entire command sequence is completed.

Flow overview:

```
Fleet Manager  -->  ROS2 Action Client  -->  Drone Action Server
      ↑                                            ↓
      └───────────  Feedback / Result Messages  ───────────┘
```

Each command received is processed sequentially, and a delay (*sleep*) is introduced to simulate realistic drone behavior. Feedback messages are sent after each command, while results are sent once all commands have been processed.

class commands_action_server. CommandsActionServer (** args : Any* , *** kwargs : Any*) [source]

Bases: `Node`

ROS2 Action Server node responsible for executing drone missions.

This class handles incoming action goals from the Fleet Manager, simulates command execution using the *process_command* function, publishes feedback messages, and returns results when the mission completes.

Parameters :

- **drone_id** (*int*) – Unique identifier of the drone instance.
- **drone_logger** (*logging.Logger*) – Logger instance used for recording drone system activity.

__init__ (*drone_id : int* , *drone_logger*) [source]

execute_callback (*goal_handle*) [source]

Executes a mission goal received from the Fleet Manager.

This method processes each command in the received mission sequentially. For each command: - It simulates execution via *process_command()* - Publishes feedback with the current command status - Waits briefly between commands to emulate real drone timing

Once all commands are executed, it publishes the final result.

Parameters :

goal_handle (*rclpy.action.server.ServerGoalHandle*) – Handle representing the received goal.

Returns :

Result message containing mission completion information.

Return type :

commands_action_interface.action.Commands.Result

Raises :

Exception – If the ROS2 system is interrupted or a command fails.

commands_action_server. run_action_server (*drone_id : int* , *drone_logger*) [source]

Starts and manages the ROS2 Action Server for a given drone.

This function initializes the ROS2 environment, spins up the action server, and handles clean shutdowns upon keyboard interruption or system stop.

Parameters :

- **drone_id** (*int*) – Unique identifier of the drone instance.
- **drone_logger** (*logging.Logger*) – Logger instance used for recording drone system activity.

Returns :

None

Commands Components

Commands Processor

This module is responsible for parsing, interpreting, and executing mission commands received by the drone’s ROS2 Action Server. It serves as the bridge between the high-level mission definitions (sent by the Fleet Manager) and the specific simulated actions implemented in the *Commands.commands_simulator* module.

Main responsibilities:

1. **Parse** incoming command JSON strings and normalize their keys.

2. **Select** the appropriate command class (e.g., *TakeOff* , *Land* , *Go_Waypoint* , etc.).

3. **Execute** the command and capture its result or error.

4. **Return** structured feedback to the calling ROS2 Action Server.

Expected command format:

Each command is expected to be a JSON-formatted string with the following structure:

```
{
    "missionId": 101,
    "commandId": 1,
    "commandType": "TAKEOFF",
    "params": {
        "altitude": 10
    }
}
```

The parser accepts both *camelCase* and *snake_case* naming conventions for compatibility.

commands_processor. process_command (*command_str : str* , *logger : Logger*) → Tuple [int , Dict] [source]

Process a command received as a JSON string, execute it, and return the result.

This function orchestrates the full command lifecycle: - Parses the input JSON string into a Python dictionary. - Selects the corresponding command class based on its type. - Executes the command using its *execute()* method. - Collects and returns the resulting data or error details.

Parameters :

- **command_str** (*str*) – Command in JSON string format (supports both camelCase and snake_case).

- **logger** (*logging.Logger*) – Logger instance used to record command execution progress and results.

Returns :

A tuple containing the execution status and the resulting data.

Return type :

Tuple[int, Dict]

Raises :

ValueError – If the input command string cannot be parsed as valid JSON.

commands_processor. parse_command (*command_str : str* , *logger : Logger*) → Dict [source]

Parse a JSON-formatted string and normalize its keys to *snake_case* .

This function allows flexible input by supporting both *camelCase* and *snake_case* naming conventions in the incoming JSON. It ensures that the resulting dictionary always contains consistent field names.

Parameters :

- **command_str** (*str*) – JSON string containing the command definition.
- **logger** (*logging.Logger*) – Logger instance used for debug or error messages.

Returns :

Normalized command dictionary with the following keys: - *mission_id* - *command_id* - *command_type* - *params*

Return type :

Dict

Raises :

ValueError – If the JSON string is invalid or cannot be parsed.

commands_processor. select_command (*command_dict : Dict* , *logger : Logger*) → Command [source]

Select the appropriate command class based on the *command_type* field.

This function maps each command type to its corresponding command class from the *Commands.commands_simulator* module. If the command type is unrecognized, it defaults to the base *Command* class.

Parameters :

- **command_dict** (*Dict*) – Dictionary containing normalized command data.
- **logger** (*logging.Logger*) – Logger instance used for logging the command type being processed.

Returns :

Instance of a command class ready to be executed.

Return type :

Command

Commands Simulator

This module simulates the execution of drone mission commands in a controlled environment. It defines a hierarchy of command classes representing possible actions a drone can perform, such as takeoff, landing, moving to a waypoint, taking pictures, analyzing images, and applying treatments.

Each command follows a consistent interface defined by the abstract base class *Command* , which enforces the implementation of two key methods: - `execute()` : Simulates the command's operation. - `result()` : Returns structured data summarizing the outcome.

This simulation layer allows for testing the drone mission workflow without requiring physical drones or real sensors.

Typical command parameters

All commands expect a set of parameters passed as a dictionary, usually containing:

- `latitude` (*float*): Target latitude coordinate.

- `longitude` (*float*): Target longitude coordinate.
- Additional fields depending on the command type (e.g., `cameraDevice` , `analyseModel` , etc.).

class commands_simulator. Command (*mission_id : int* , *command_id : int* , *command_type : str* , *logger : Logger*) [source]

Bases: `ABC`

Abstract base class for all simulated drone commands.

Defines the common interface used by all command types and provides access to basic metadata and logging.

Parameters :

- **mission_id** (*int*) – Mission identifier associated with this command.
- **command_id** (*int*) – Unique identifier of this command within the mission.
- **command_type** (*str*) – Command type string (e.g., “TAKEOFF”, “LAND”).
- **logger** (*logging.Logger*) – Logger instance for structured message output.

__init__ (*mission_id : int* , *command_id : int* , *command_type : str* , *logger : Logger*) [source]

abstract execute ( ) [source]

Execute the command simulation.

abstract result ( ) [source]

Return a structured dictionary with the command results.

class commands_simulator. TakeOff (*mission_id* , *command_id* , *command_type* , *params* , *logger*) [source]

Bases: `Command`

Simulates a drone takeoff operation.

Parameters :

params (*dict*) – Dictionary containing ‘latitude’ and ‘longitude’ keys.

__init__ (*mission_id* , *command_id* , *command_type* , *params* , *logger*) [source] execute ( ) [source]

Simulates the drone ascending into flight.

result ( ) [source]

Returns confirmation of a successful takeoff.

class commands_simulator. Land (*mission_id* , *command_id* , *command_type* , *params* , *logger*) [source]

Bases: `Command`

Simulates a drone landing operation.

Parameters :

params (*dict*) – Dictionary containing ‘latitude’ and ‘longitude’ keys.

__init__ (*mission_id* , *command_id* , *command_type* , *params* , *logger*) [source]

execute ( ) [source]

Simulates the drone descending and landing.

result ( ) [source]

Returns confirmation of a successful landing.

class commands_simulator. Go_Waypoint (*mission_id* , *command_id* , *command_type* , *params* , *logger*) [source]

Bases: `Command`

Simulates movement to a specific waypoint.

Parameters :

params (*dict*) – Dictionary containing ‘latitude’ and ‘longitude’ keys.

__init__ (*mission_id* , *command_id* , *command_type* , *params* , *logger*) [source]

execute ( ) [source]

Simulates travel to the target waypoint.

result ( ) [source]

Returns confirmation that the waypoint has been reached.

class commands_simulator. Go_Home (*mission_id* , *command_id* , *command_type* , *params* , *logger*) [source]

Bases: `Command`

Simulates the drone returning to its home base.

Parameters :

params (*dict*) – Dictionary containing 'latitude' and 'longitude' keys.

__init__ (*mission_id* , *command_id* , *command_type* , *params* , *logger*) [source]

execute ( ) [source]

Simulates the drone flying back to its home location.

result ( ) [source]

Returns confirmation of a successful return.

class commands_simulator. Take_Picture (*mission_id* , *command_id* , *command_type* , *params* , *logger*) [source]

Bases: `Command`

Simulates taking a picture using the drone's camera.

Parameters :

params (*dict*) – Dictionary containing 'latitude', 'longitude', and 'cameraDevice' keys.

__init__ (*mission_id* , *command_id* , *command_type* , *params* , *logger*) [source]

execute ( ) [source]

Simulates image capture from the drone camera.

Attempts to load a predefined image from the *Data/* directory based on the current coordinates.

result ( ) [source]

Returns a base64-encoded image string along with metadata.

class commands_simulator. Analyse_Image (*mission_id* , *command_id* , *command_type* , *params* , *logger*) [source]

Bases: `Command`

Simulates image analysis using a preloaded model or mock dataset.

Parameters :

params (*dict*) – Dictionary containing 'latitude', 'longitude', and 'analyseModel' keys.

__init__ (*mission_id* , *command_id* , *command_type* , *params* , *logger*) [source]

execute ( ) [source]

Simulates image processing and analysis.

Loads a mock result from *Data/amounts.json* and an analyzed image from *Data/<lat>_<lon>_analysed.png* if available.

result ( ) [source]

Returns analysis results and the processed image (if available).

class commands_simulator. Treatment (*mission_id* , *command_id* , *command_type* , *params* , *logger*) [source]

Bases: `Command`

Simulates the application of an agricultural treatment.

Parameters :

params (*dict*) – Dictionary containing ‘latitude’, ‘longitude’, ‘treatment’, and ‘amount’ keys.

__init__ (*mission_id* , *command_id* , *command_type* , *params* , *logger*) [source]

execute ( ) [source]

Simulates spraying or applying a treatment at a target location.

result ( ) [source]

Returns confirmation of treatment application.