Lab04: Companion Computer – Autopilot Integration

Disclaimer and Usage Notes

These course materials span over 100 pages and contain extensive technical documentation, detailed code samples, architectural diagrams, and comprehensive command references. This extensive scope is **intentionally designed** and should not be viewed as content to be covered in a single 2-4 hour laboratory session.

Educational Philosophy

Autonomous maritime systems represent a vast and rapidly evolving field that encompasses multiple disciplines: embedded systems, network communication, control theory, computer vision, maritime engineering, and safety protocols. Rather than providing superficial coverage, these materials serve as a **comprehensive foundation resource** that you can reference, revisit, and build upon throughout your academic development.

The progressive project structure takes you on a systematic journey from fundamental concepts - basic heartbeat printouts and MQTT communication - through to more sophisticated autonomous multi-vessel coordination systems. Working with the SITL environment established in Lab03, you'll develop increasingly complex Python implementations that build upon each previous project's functionality.

We hope that this approach provides both practical experience with waterborne autonomous systems and a **solid conceptual foundation**, empowering you to confidently tackle advanced research topics in maritime informatics.

Recommended Usage Approach

These materials function as **living documentation** rather than traditional lecture slides. You should:

- Use specific sections as needed for current projects and assignments
- Return to foundational concepts when tackling advanced implementations
- Leverage the comprehensive code examples as starting points for your own development
- Reference the troubleshooting sections and architectural patterns during independent work

Future Repository Integration

These materials may eventually link to **an organized GitHub repository** containing complete project implementations, additional examples, and our SmartMove Lab's open source contributed extensions.

Bottom Line: Consider these materials a comprehensive reference manual for maritime autonomy rather than a single-session curriculum - they're designed to support your learning journey from initial concepts through advanced independent research.

Table Of Contents:

1. Introduction and Objectives

1.1 From Simulation to Autonomous Control

In Lab03, you mastered Software In The Loop (SITL) simulation and learned to control ArduPilot vehicles through MAVProxy commands and pre-programmed missions. Now, Lab04 introduces the next critical component of autonomous maritime systems: **companion computer integration**.

A companion computer **transforms an autopilot from a mission executor into an intelligent autonomous system**. While the autopilot handles low-level control (maintaining heading, following waypoints, stabilizing the vessel), the companion computer provides high-level intelligence: dynamic decision-making, inter-vessel coordination, and adaptive behavior based on environmental conditions.

1.2 The Companion Computer Architecture

In our maritime robotics ecosystem, each vessel employs:

- **Autopilot** (Pixhawk/CUAV): Executes navigation commands, maintains vessel stability
- **Raspberry Pi 5 (8GB RAM)**: Runs Python scripts for autonomous behavior and communication
    - Quad-core Arm Cortex-A76 @ 2.4GHz processor
    - 8GB LPDDR4X-4267 SDRAM for complex processing tasks
    - High-performance SD card (SanDisk Ultra microSDXC 128GB) for reliable storage
    - Latest Raspberry Pi OS (64-bit) for optimal performance
    - Dual-band WiFi and Gigabit Ethernet for flexible connectivity
- **4G Connectivity**: Enables real-time coordination via MQTT protocols
- **Shore Station**: Monitors and coordinates fleet operations

This architecture separates concerns effectively:

- **Autopilot firmware** remains focused on reliable vehicle control
- **Python scripts** implement mission-specific logic without modifying firmware
- **MQTT communication** enables scalable multi-vessel coordination
- **Modular design** allows rapid development and testing of new behaviors

The Raspberry Pi 5's significant performance improvements over previous generations ensure smooth operation of computer vision algorithms, real-time communication, and complex autonomous behaviors even in challenging maritime environments.

1.3 Lab Objectives

By the end of this lab, you will be able to:

1. **Establish Communication** between Python scripts and ArduPilot (via DroneKit/PyMavlink)

2. **Read Telemetry Data** including position, heading, speed, and system status
3. **Control Vehicle Behavior** through mode changes and movement commands
4. **Upload and Monitor Missions** programmatically via Python
5. **Implement MQTT Communication** for vessel-to-vessel and shore-to-vessel coordination
6. **Develop Autonomous Behaviors** such as follow-the-leader and coordinated operations
7. **Transition from SITL to Hardware** using proper serial connections and device management

1.4 Project Progression Overview

The lab follows a carefully designed progression through 8 practical projects:

- **Projects 1-3**: Foundation - Basic connectivity, telemetry, and modular design
- **Projects 4-5**: Multi-vehicle - Role-based control and JSON communication
- **Project 6**: Autonomous behaviour - Command processing and follow-the-leader (conceptual)
- **Project 7**: Autonomous behaviour - Actual Implementation of follow-the-leader scenario (full implementation)
- **Project 8**: Mission management - Dynamic waypoint upload and monitoring (conceptual)

Each project builds upon previous concepts while introducing new capabilities.

1.5 Real-World Applications

The skills developed in this lab directly apply to:

- **Autonomous Survey Operations**: Vessels dynamically adjusting survey patterns
- **Coordinated Search Patterns**: Multiple vessels covering areas efficiently
- **Convoy Operations**: Automated following for fuel efficiency and safety

- **Dynamic Mission Adaptation**: Responding to weather changes or obstacles
- **Fleet Management**: Shore-based monitoring and control of multiple vessels

1.6 Development Approach

This lab emphasizes practical learning through incremental development:

1. **SITL First**: All concepts are tested in simulation before hardware deployment
2. **Modular Code**: Object-oriented design for maintainability and reusability
3. **Progressive Complexity**: Each project adds one key concept at a time

---

2. Development Environment Setup

2.1 Python Virtual Environment

A Python virtual environment isolates project dependencies, preventing conflicts between different Python projects and ensuring reproducible installations. This is essential when working with specific library versions like DroneKit.

Step 1: Navigate to your working directory

Create a dedicated directory structure for Lab04 projects:

```
# Create main directory for all Lab04 work
mkdir -p ~/lab04-companion
cd ~/lab04-companion
```

# Create subdirectory for code projects

```
mkdir code
cd code
```

Step 2: Create the virtual environment

# Create a new virtual environment named 'ss_venv' (Summer School Virtual Environment)

```
python3 -m venv ss_venv
```

# You should now see a new directory called ss_venv/

```
ls -la
# Expected output includes: ss_venv/
```

Step 3: Activate the virtual environment

```
# For Linux/WSL2/Ubuntu
source ss_venv/bin/activate

# You should see (ss_venv) prefix in your terminal prompt
# Example: (ss_venv) user@computer:~/lab04-companion/code$
```

Step 4: Verify activation

```
# Check that pip points to the virtual environment
which pip
# Should output:
/home/<username>/lab04-companion/code/ss_venv/bin/pip

# Check Python version
python --version
# Should show your Python 3.x version
```

Important Virtual Environment Commands (not needed - for possible troubleshooting only):

```
# To deactivate the virtual environment (when done working)
deactivate

# To reactivate later (from the code directory)
source ss_venv/bin/activate

# To delete the virtual environment completely (if needed)
# First deactivate, then:
rm -rf ss_venv/
```

Project Directory Structure Preview:

After setup, your directory structure will look like:

```
~/lab04-companion/
└── code/                # Parent directory for all projects
    ├── ss_venv/         # Virtual environment (created above)
    ├── .env             # Environment variables (will create next)
    ├── ca.crt           # CA certificate for TLS (instructor-provided)
    ├── requirements.txt # Python dependencies (will create next)
    ├── 1_proj/          # Project 1 files
    ├── 2_proj/          # Project 2 files
    └── ...              # Additional projects
```

2.3 Establishing Secure Connections to SmartMove MQTT Broker

Critical - make sure you follow next steps to receive the correct ca.crt file which is absolutely needed in order to establish secure connections to the SmartMove Mosquitto MQTT broker which we will use in future projects in this lab.

```
# Make sure you are inside your 'code' directory:
cd ~/lab04-companion/code
pwd

# should show something like: /home/thomas/lab04-companion/code

# Then, run the following command:
curl -o ca.crt https://smartmove.aegean.gr/certs/ca.crt

# you just downloaded the ca.crt file (public certificate)
# verify it by:
file ca.crt

# you should see:
ca.crt: PEM certificate
```

Context:

The `ca.crt` file you have just downloaded is the public root certificate for the well-known authority **Let's Encrypt**, which we are using as our system's Certificate Authority (CA). This file acts as the "trust anchor" for all secure communication. Shortly, our Python scripts will establish a secure connection to our Mosquitto broker at `smartmove-local.syros.aegean.gr.` During that connection attempt, the broker will present its own server certificate to prove its identity. Our Python client will then use this `ca.crt` file to verify that the broker's certificate is authentic and was indeed issued by Let's Encrypt, ensuring we are not connecting to a malicious imposter and securing our data with TLS encryption.

Note for Windows Users: If using Windows with WSL2 (as set up in Lab03), all these commands should be run in the Ubuntu terminal, not Windows Command Prompt or PowerShell.

2.3 General Troubleshooting Guide: Using Files from Windows/GitHub in WSL (Optional but helpful)

When moving text-based files (like `.crt`, `.pem`, `.conf`, `.sh`) from Windows or a web browser into the Windows Subsystem for Linux (WSL), you may encounter issues with file format or line endings. Here's a quick guide to diagnose and fix them.

1. Diagnose the File

Before using a copied file, always check its integrity first. The `file` command is the best tool for this. To check a single file or multiple files with the same extension:

```
# Check a single file
file your_file_name.crt
```

Based on the output, proceed to the relevant solution below.

2. Issue: Incorrect File Format (e.g., `HTML document`)

This is common when (for example) downloading files from GitHub's web interface.

- **Symptom:** `file your_file.crt` outputs `HTML document...`
- **Cause:** You saved the GitHub page that displays the file, not the raw file content itself.
- **Solution:** Download the raw file content directly into WSL.
  1. On the file's GitHub page, click the **"Raw"** button.
  2. Copy the URL from your browser's address bar.

3. Issue: Incorrect Line Endings (Windows CRLF)

- **Symptom:** `file your_file.txt` outputs `...with CRLF line terminators.` Scripts or applications using the file may fail.
- **Cause:** Windows uses a different line ending format (`\r\n`) than Linux (`\n`).
- **Solution:** Convert the file using the `dos2unix` utility.

Install `dos2unix` (if you don't have it):

```
sudo apt update && sudo apt install dos2unix
```

Run the conversion (this modifies the file in place):

```
dos2unix your_file.crt
```

4. Issue: Alternate Data Streams (`:Zone.Identifier`)

- **Symptom:** You see a duplicate file with `:Zone.Identifier` at the end (e.g., `ca.crt:Zone.Identifier`).
- **Cause:** Windows adds this metadata to files downloaded from the internet. WSL displays it as a separate file.
- **Solution:** This metadata file is not needed in Linux and is safe to delete.

```
rm 'ca.crt:Zone.Identifier'
```

2.2 Installing Dependencies

Our projects require several Python libraries for autopilot communication and MQTT messaging. We'll use a `requirements.txt` file to manage these dependencies.

Step 1: Create requirements.txt

In the `code` directory (with virtual environment activated):

```
# Ensure you're in the correct directory
pwd
# Should show: /home/<username>/lab04-companion/code

# Create requirements.txt file
nano requirements.txt

Add the following content:
dronekit==2.9.2
future==1.0.0
lxml==5.3.0
monotonic==1.6
paho-mqtt==2.1.0
pymavlink==2.4.41
python-dotenv==1.0.1
```

Save and exit (Ctrl-O, then Ctrl+X).

Step 2: Install dependencies

```
# Install all required packages
pip install -r requirements.txt

# This will install:
# - dronekit: High-level API for ArduPilot communication
# - paho-mqtt: MQTT client for vessel coordination
# - python-dotenv: Environment variable management
# - Plus all their dependencies (future, lxml, monotonic, pymavlink)
```

Step 3: Verify installation

```
# List installed packages
pip list

# Verify specific packages
python -c "import dronekit; print(f'DroneKit version: {dronekit.__version__}')"
python -c "import paho.mqtt; print('MQTT client imported successfully')"
python -c "import dotenv; print('python-dotenv imported successfully')"
```

Library Explanations:

- **DroneKit (2.9.2)**: Provides a high-level Python API for communicating with ArduPilot. Although no longer actively maintained, it remains functional and beginner-friendly for learning autopilot integration.

- **paho-mqtt (2.1.0)**: The Eclipse Paho MQTT Python client enables publish/subscribe messaging between vessels and shore stations. Essential for multi-vessel coordination.

- **python-dotenv (1.0.1)**: Loads environment variables from `.env` files, keeping sensitive configuration (passwords, connection strings) separate from code.
- **pymavlink (2.4.41)**: Low-level MAVLink protocol implementation that DroneKit uses internally. Can be used directly for advanced operations.
- **Supporting libraries**:
  - `future:` Python 2/3 compatibility
  - `lxml:` XML processing for MAVLink
  - `monotonic:` Time-related functions

Common Installation Issues:

If you encounter errors during installation:

```
# Update pip first
pip install --upgrade pip

# If specific package fails, try installing individually
pip install dronekit==2.9.2 --no-deps
pip install pymavlink==2.4.41
# Then retry full requirements.txt

# For permission errors (should not happen in venv)
pip install --user -r requirements.txt
```

2.3 Critical: DroneKit Compatibility Fix

DroneKit was last updated in 2019 and has a compatibility issue with Python 3.10 and newer versions. When you first run a script using DroneKit, you'll encounter an error. This section explains the issue and provides the fix.

The Error You'll Encounter:

When running your first DroneKit script, you'll see:

`AttributeError: module 'collections' has no attribute 'MutableMapping'`

This occurs because Python 3.10 moved `MutableMapping` from `collections` to `collections.abc.`

Step 1: Attempt to import DroneKit (to see the error)

```
# With virtual environment activated
python -c "import dronekit"

# You'll see an error traceback ending with:
# File ".../dronekit/__init__.py", line 2689, in <module>
#   class Parameters(collections.MutableMapping, HasObservers):
# AttributeError: module 'collections' has no attribute 'MutableMapping'
```

Step 2: Locate the DroneKit installation

```
# Find the exact path to DroneKit
python -c "import site; print(site.getsitepackages()[0])"

# This will output something like:
# /home/<username>/lab04-companion/code/ss_venv/lib/python3.10/site-packages

# Navigate to DroneKit directory
cd ss_venv/lib/python3.*/site-packages/dronekit/
```

Step 3: Fix the compatibility issue

```
# Open the __init__.py file for editing
nano __init__.py

# Search for the problematic line (Ctrl+W in nano)
# Search for: MutableMapping
# You'll find around line 2689:
```

```
# class Parameters(collections.MutableMapping, HasObservers):
```

Change this line from:

`class Parameters(collections.MutableMapping, HasObservers):`

To:

`class Parameters(collections.abc.MutableMapping, HasObservers):`

Save and exit (Ctrl+O, then Ctrl+X, then Enter).

Step 4: Verify the fix

```
# Return to the code directory
cd ~/lab04-companion/code

# Test DroneKit import again
python -c 'import dronekit; print("DroneKit imported successfully!")'
# Should output: DroneKit imported successfully!
```

Understanding the Fix:

- **collections.MutableMapping** was deprecated in Python 3.3
- It was moved to **collections.abc.MutableMapping**
- Python 3.10 finally removed the old location
- Our fix simply updates DroneKit to use the new location

Alternative Solutions (if manual editing fails):

```
# Option 1: Use sed to automatically make the change
```

```
cd ss_venv/lib/python3.*/site-packages/dronekit/
sed -i 's/collections.MutableMapping/collections.abc.MutableMapping/g' __init__.py

# Option 2: Use a patched fork (if available - could not find one thus far)
# pip uninstall dronekit
# pip install git+https://github.com/[patched-fork-url]
```

Important Notes:

1. This fix must be applied each time you create a new virtual environment with DroneKit
2. The issue only affects Python 3.10+; Python 3.9 and earlier work without modification
3. Despite being unmaintained, DroneKit remains functional for learning purposes
4. For production systems, consider migrating to PyMAVLink or MAVSDK-Python

Why We Still Use DroneKit:

Despite its maintenance status, DroneKit offers:

- Beginner-friendly API design
- Excellent abstraction of MAVLink complexity
- Comprehensive documentation (though dated)
- Quick prototyping capabilities
- Easy transition to understanding PyMAVLink later

The fix is simple and one-time per environment, making DroneKit still viable for educational purposes.

2.4 Environment Variables (.env)

Environment variables keep sensitive configuration (credentials, connection strings) separate from code. We use python-dotenv to load these variables from a `.env` file.

Important Note: Single .env for All Projects

We use a single `.env` file shared across all projects (1-7) for simplicity. This means:

- Some variables won't be used in early projects
- Later projects use role-based variables (SCOUT_*, TEAM1_*, etc.)
- This approach mirrors real-world systems where configurations often contain unused variables
- Clear comments indicate which projects use which variables

Step 1: Create the .env file

In the `code` directory (where requirements.txt is located):

```
# Ensure you're in the correct directory
pwd
# Should show: /home/<username>/lab04-companion/code

# Create .env file
nano .env
```

Step 2: Add complete configuration

The Maritime Informatics & Robotics Summer School uses a dedicated Mosquitto MQTT broker hosted at the University of the Aegean in Syros. This broker (`smartmove-local.syros.aegean.gr`) is configured to support both secure TLS and standard connections, providing a reliable communication platform for our maritime robotics projects.

Copy this entire configuration. Comments indicate which projects use each section:

```
# ===== Lab04 Companion Computer Configuration =====
# This single .env file is used by all projects (1-7)
# Not all variables are used in every project - see comments

# ===== CORE MQTT SETTINGS (Projects 2-7) =====
# These are essential for any project using MQTT
```

```
MQTT_BROKER=smartmove-local.syros.aegean.gr
MQTT_PORT=1883                      # Use 8883 for TLS
MQTT_USE_TLS=false                 # Set to true for secure connection
MQTT_CA_CERT_PATH=ca.crt           # Only needed when MQTT_USE_TLS=true

# ===== BASIC MQTT CREDENTIALS (Projects 2-3) =====
# Projects 2-3 use these simple credentials
# Note: Projects 4-7 use role-specific credentials below instead

MQTT_USERNAME=scout                # Default username for projects 2-3
MQTT_PASSWORD=scout                # Default password for projects 2-3

# ===== MQTT TOPICS (Projects 4-7) =====
# Topics for multi-vessel coordination

SCOUT_POSITION_TOPIC=scout/position
TEAM1_POSITION_TOPIC=team1/position
TEAM2_POSITION_TOPIC=team2/position
TEAM3_POSITION_TOPIC=team3/position

# ===== ROLE-BASED MQTT CREDENTIALS (Projects 4-7) =====
# Each role has separate credentials for multi-vessel scenarios

# Scout configuration
SCOUT_MQTT_USERNAME=scout
SCOUT_MQTT_PASSWORD=scout

# Team 1 configuration
TEAM1_MQTT_USERNAME=team1
TEAM1_MQTT_PASSWORD=team1
TEAM1_COMMANDS=team1/commands

# Team 2 configuration
TEAM2_MQTT_USERNAME=team2
TEAM2_MQTT_PASSWORD=team2
TEAM2_COMMANDS=team2/commands

# Team 3 configuration
```

```
TEAM3_MQTT_USERNAME=team3
TEAM3_MQTT_PASSWORD=team3
TEAM3_COMMANDS=team3/commands

# ===== SITL CONNECTION STRINGS (Projects 4-7) =====
# UDP connections for SITL simulation
# Projects 1-3 use hardcoded connection strings

SCOUT_CONNECTION_STRING=udp:127.0.0.1:14550
TEAM1_CONNECTION_STRING=udp:127.0.0.1:14560
TEAM2_CONNECTION_STRING=udp:127.0.0.1:14570
TEAM3_CONNECTION_STRING=udp:127.0.0.1:14580

# ===== HARDWARE CONNECTION STRINGS (Future use) =====
# Serial connections for real hardware (covered in Section 4)
# Currently commented out - uncomment when using real hardware

# SCOUT_REAL_CONNECTION_STRING=/dev/cuav
# TEAM1_REAL_CONNECTION_STRING=/dev/pixhawk1
# TEAM2_REAL_CONNECTION_STRING=/dev/pixhawk2
# TEAM3_REAL_CONNECTION_STRING=/dev/pixhawk3

# ===== OPERATIONAL PARAMETERS (Projects 7+) =====
# Behavioral parameters for autonomous operations

SAFE_FOLLOW_DISTANCE=10          # Minimum distance in meters
TELEMETRY_INTERVAL=1             # Telemetry publish rate in seconds
```

Step 3: Configure for TLS (Optional but Recommended)

For secure MQTT connections, modify these lines:

```
MQTT_PORT=8883                    # TLS port
MQTT_USE_TLS=true                # Enable TLS
MQTT_CA_CERT_PATH=ca.crt        # Path to certificate
```

And ensure the CA certificate exists:

```
# We will provide ca.crt file
# Place it in the code directory (same location as .env)
ls -la ca.crt
# Should show the certificate file
```

Step 4: Understanding Variable Usage by Project

Project	Variables Used	Notes
1	None	Hardcoded connection string
2	MQTT_BROKER, MQTT_PORT, MQTT_USERNAME, MQTT_PASSWORD	Basic MQTT
3	Same as Project 2	Modular design
4	All role-based variables (SCOUT_*, TEAM_*)	Multi-vessel
5-7	Same as Project 4	Advanced features

Step 5: Security Considerations (not applicable now, but explain best practices)

```
# Ensure .env is NOT tracked by git
echo ".env" >> .gitignore
echo "ca.crt" >> .gitignore     # optional
```

```
# Check file permissions (optional)
chmod 600 .env  # Only owner can read/write
```

Testing Environment Variables:

Create a test script to verify your configuration:

# test_env.py

```python
from dotenv import load_dotenv
import os

# Load environment variables
load_dotenv()

# Test core MQTT settings
print("=== CORE MQTT SETTINGS ===")
print(f"Broker: {os.getenv('MQTT_BROKER')}")
print(f"Port: {os.getenv('MQTT_PORT')}")
print(f"Using TLS: {os.getenv('MQTT_USE_TLS')}")

# Test role-based settings
print("\n=== ROLE-BASED SETTINGS ===")
print(f"Scout Username: {os.getenv('SCOUT_MQTT_USERNAME')}")
print(f"Team1 Connection: {os.getenv('TEAM1_CONNECTION_STRING')}")
```

Run the test:

```
python test_env.py
```

```
# Expected output:
# === CORE MQTT SETTINGS ===
# Broker: smartmove-local.syros.aegean.gr
# Port: 1883
# Using TLS: false
```

Why Single .env?

1. **Simplicity**: Configure once, use everywhere
2. **Maintenance**: Update broker settings in one place
3. **Learning**: See how configuration grows with project complexity
4. **Realistic**: Production systems often have unused configuration variables

Important Notes:

- Early projects ignore extra variables (no harm done)
- Comment/uncomment sections as needed for testing
- When moving to hardware, update the *_REAL_CONNECTION_STRING variables
- Each team should coordinate to avoid credential conflicts while testing

2.5 MQTT Broker Configuration

Before diving into projects, let's verify our MQTT broker connection is working correctly. This section tests both the basic connection and TLS configuration.

Understanding the University of Aegean MQTT Broker

The `smartmove-local.syros.aegean.gr` broker:

- Supports both standard (port 1883) and TLS (port 8883) connections
- Requires authentication (no anonymous access)
- Provides isolated topic namespaces for different vessels/students
- Maintained specifically for maritime robotics research and education

Step 1: Install MQTT command-line tools (optional but helpful)

```
# Install mosquitto clients for testing
sudo apt update
sudo apt install mosquitto-clients -y
```

Step 2: Test basic MQTT connectivity

[Run these tests in discrete terminals]

Let’s test the non-TLS connection first

In one terminal (subscribe):

```
# Test subscribe
mosquitto_sub -h smartmove-local.syros.aegean.gr -p 1883 -u asv1 -P asv1 -t test/connection -v
```

In another terminal (publish):

```
# Test publish (you can replace with your team's asv number)
mosquitto_pub -h smartmove-local.syros.aegean.gr -p 1883 -u asv1 -P asv1 -t test/connection -m "Hello from Lab04"
```

If all goes well, in the first terminal where you have subscribed, you should see:

```
test/connection Hello from Lab04
```

Step 3: Create Python MQTT test script

Create a more comprehensive test that matches our project structure:

```
# test_mqtt.py
import paho.mqtt.client as mqtt
import os
from dotenv import load_dotenv
import time

# Load environment variables
```

```python
load_dotenv()

# MQTT settings from .env
broker = os.getenv('MQTT_BROKER')
port = int(os.getenv('MQTT_PORT'))
username = os.getenv('MQTT_USERNAME')
password = os.getenv('MQTT_PASSWORD')

# Callback functions
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print(f"Connected successfully to {broker}:{port}")
        # Subscribe to test topic
        client.subscribe("test/lab04")
    else:
        print(f"Connection failed with code {rc}")

def on_message(client, userdata, msg):
    print(f"Received: {msg.topic} -> {msg.payload.decode()}")

# Create MQTT client
client = mqtt.Client()
client.username_pw_set(username, password)
client.on_connect = on_connect
client.on_message = on_message

# Connect and start loop
print(f"Connecting to {broker}:{port} as {username}...")
client.connect(broker, port, 60)
client.loop_start()

# Publish test messages
time.sleep(2)  # Wait for connection
for i in range(3):
    message = f"Test message {i+1} from Lab04"
    client.publish("test/lab04", message)
    print(f"Published: {message}")
    time.sleep(1)
```

```
# Keep running for a bit to receive messages
time.sleep(5)
client.loop_stop()
client.disconnect()
```

Run the test:

```
python test_mqtt.py
```

```
# Expected output:
# Connecting to smartmove-local.syros.aegean.gr:1883 as scout...
# Connected successfully to smartmove-local.syros.aegean.gr:1883
# Published: Test message 1 from Lab04
# Received: test/lab04 -> Test message 1 from Lab04
# Published: Test message 2 from Lab04
# Received: test/lab04 -> Test message 2 from Lab04
# Published: Test message 3 from Lab04
# Received: test/lab04 -> Test message 3 from Lab04
```

Step 4: Test TLS connection (optional)

To test secure TLS connection:

1. Ensure you have the CA certificate:

```
ls -la ca.crt
# File should exist in code directory
```

2. Update .env for TLS:

```
MQTT_PORT=8883
MQTT_USE_TLS=true
```

3. Each team should put their respective credentials (asv1/asv1, asv2/asv2 etc)

```
MQTT_USERNAME=asv1                # Default username for initial tests (assuming asv1 team)
MQTT_PASSWORD=asv1                # Default password for initial tests (assuming asv1 team)
```

4. Create TLS test script:

   test_mqtt_secure.py:

```python
import ssl
import os
import paho.mqtt.client as mqtt
from dotenv import load_dotenv
import time

load_dotenv()

broker   = os.getenv("MQTT_BROKER")
port     = int(os.getenv("MQTT_PORT", 8883))
username = os.getenv("MQTT_USERNAME")
password = os.getenv("MQTT_PASSWORD")
use_tls  = os.getenv("MQTT_USE_TLS", "false").lower() == "true"

def on_connect(client, userdata, flags, rc):
    print("Connected" if rc==0 else f"Connect failed ({rc})")
    client.subscribe("test/lab04")

def on_message(client, userdata, msg):
    print(f"Received: {msg.topic} -> {msg.payload.decode()}")

client = mqtt.Client()
client.username_pw_set(username, password)
client.on_connect = on_connect
client.on_message = on_message

if use_tls:
    # create an SSLContext that loads system CAs (which include Let's Encrypt)
    ctx = ssl.create_default_context()
    client.tls_set_context(ctx)
    print("TLS enabled, using system default CA certificates")

print(f"Connecting to {broker}:{port} ...")
client.connect(broker, port, keepalive=60)
client.loop_start()

time.sleep(2)
for i in range(3):
    msg = f"Test message {i+1}"
    client.publish("test/lab04", msg)
    print("Published:", msg)
    time.sleep(1)

time.sleep(5)
client.loop_stop()
client.disconnect()
```

You should see an output like:

```
TLS enabled, using system default CA certificates
Connecting to smartmove-local.syros.aegean.gr:8883 ...
Connected
Published: Test message 1
Received: test/lab04 -> Test message 1
Published: Test message 2
Received: test/lab04 -> Test message 2
Published: Test message 3
Received: test/lab04 -> Test message 3
```

Common Connection Issues:

1. **Connection refused**: Check firewall, ensure broker is accessible
2. **Authentication failed**: Verify username/password in .env
3. **TLS certificate error**: Ensure ca.crt exists and is valid

4. **Network timeout**: Check network connectivity to broker

Best Practices:

- Test connectivity before running complex projects
- Use meaningful topic names for your tests
- Clean up test topics when done
- Monitor broker logs if you have access

Once these tests pass, you're ready to proceed with the projects!

---

3. SITL-Based Python Development

This section guides teams through progressive Python projects that build autonomous vessel control capabilities. Each of the 8 teams will work with their assigned hardware kit, which includes:

- Custom-built boat with thrusters and ESCs
- Pixhawk 2.4.8 PX4 Flight Controller running ArduPilot Rover (FRAME_CLASS=2)
- Raspberry Pi 5 (8GB) companion computer
- Communication hardware (antennas, 4G dongles)

All initial development uses SITL simulation, allowing teams to test code safely before deploying to their physical vessels.

3.1 Project 1: Basic Connection and Telemetry

Learning Objectives:

- Establish DroneKit connection to autopilot
- Read basic telemetry data (heading, speed, position)
- Understand the MAVLink communication flow
- Create foundation for more complex projects

Project Structure:

Create the project directory:

```
# From the code directory (with venv activated)
cd ~/lab04-companion/code
mkdir 1_proj
cd 1_proj
```

Create the main script:

(*) - You could either create the file with 'touch leader_telemetry.py' or directly through VS Code, which is probably more convenient (CTRL-Shift-P -> WSL -> ...)

`leader_telemetry.py`

```python
from dronekit import connect, VehicleMode
import time
from datetime import datetime

# Connect to the vehicle (SITL) using the specified connection string
connection_string = 'udp:127.0.0.1:14550'
print(f"Connecting to vehicle on: {connection_string}")

# Establish a connection to the vehicle
# 'wait_ready=True' ensures the script waits until all vehicle parameters are loaded
vehicle = connect(connection_string, wait_ready=True)

# Function to output basic telemetry data
def get_telemetry():
    # Get the current timestamp
    timestamp = datetime.now().strftime("%m/%d/%Y - %H:%M:%S")

    # Get current heading (yaw) in degrees from the vehicle
    heading = vehicle.heading

    # Get ground speed in meters per second and round to two decimal places
    ground_speed = round(vehicle.groundspeed, 2)

    # Get latitude and longitude from the vehicle's GPS location
    lat = vehicle.location.global_frame.lat
    lon = vehicle.location.global_frame.lon

    # Print the telemetry data to the console
    print(f"Timestamp: {timestamp}")
    print(f"Heading: {heading} degrees")
    print(f"Ground Speed: {ground_speed} m/s")
    print(f"Latitude: {lat}")
    print(f"Longitude: {lon}")
    print('-' * 30)

try:
    # Main loop to continuously read and print telemetry data every 5 seconds
    while True:
        get_telemetry()  # Call the function to print telemetry data
        time.sleep(5)    # Wait for 5 seconds before the next update

except KeyboardInterrupt:
    # Catch keyboard interrupt (Ctrl+C) to exit gracefully
    print("Exiting script...")

finally:
    # Ensure the vehicle connection is closed before exiting
    vehicle.close()
    print("Vehicle connection closed.")
```

Understanding the Code:

1. **Connection String**: `udp:127.0.0.1:14550` connects to SITL on the default port
2. **wait_ready=True**: Ensures all parameters are loaded before proceeding
3. **Telemetry Access**: DroneKit provides direct attributes like `vehicle.heading`
4. **Clean Exit**: Proper connection closure prevents resource leaks

Running the Project:

Terminal 1 - Start SITL (from Lab03):

```
cd ~/sitl-test
sim_vehicle.py -v Rover -L syros \
--add-param-file=/home/<username>/custom-parms/boat.parm \
--out=udp:127.0.0.1:14550 --console --map
```

Terminal 2 - Run the Python script:

```
cd ~/lab04-companion/code
source ss_venv/bin/activate
cd 1_proj
python leader_telemetry.py
```

Expected Output:

```
Connecting to vehicle on: udp:127.0.0.1:14550
Timestamp: 06/23/2025 - 10:15:32
Heading: 180 degrees
Ground Speed: 0.02 m/s
Latitude: 37.438788
Longitude: 24.945544
------------------------------
```

Testing Vehicle Movement:

```
In the MAVProxy console (Terminal 1):
# Arm the vehicle
```

```
arm throttle

# Set to GUIDED mode
mode GUIDED
```

Then right-click on the map and select "Fly to here" to command movement (issue altitude: 0m).

Watch the telemetry update in Terminal 2. You should see updates like:

```
Timestamp: 06/23/2025 - 10:16:05
Heading: 165 degrees
Ground Speed: 5.03 m/s
Latitude: 37.438321
Longitude: 24.945789
```

Key Observations:

1. **Real-time Updates**: Telemetry reflects vehicle state changes immediately
2. **Default Speed**: Boats move at ~5 m/s in GUIDED mode (configurable)
3. **GPS Precision**: Coordinates show realistic maritime navigation precision

Common Issues and Solutions:

Issue	Solution
Connection refused	Ensure SITL is running with `--out=udp:127.0.0.1:14550`
Import error	Check virtual environment is activated

No telemetry updates	Verify vehicle is armed and receiving commands

Team Exercises:

1. **Documentation Research**: Find the official DroneKit documentation and identify additional vehicle attributes you could monitor
2. **Experiment with Timing**: Test different update rates (0.1s, 1s, 10s) and observe the output

Understanding MAVLink:

Behind the scenes, DroneKit uses MAVLink protocol to communicate:

- **Heartbeat messages** maintain connection status
- **GLOBAL_POSITION_INT** provides GPS coordinates
- **VFR_HUD** supplies heading and ground speed
- **SYS_STATUS** reports battery and system health

Preparing for Project 2:

This basic telemetry forms the foundation for:

- Publishing data to MQTT (Project 2)
- Multi-vessel coordination (Projects 4-7)
- Mission monitoring (Project 8)

Code Summary:

The complete project demonstrates:

- Minimal code for maximum functionality
- Clean structure suitable for expansion
- Error handling for production readiness

- Real-time data access for autonomous decisions

Teams should ensure this works reliably before proceeding, as all subsequent projects build upon this connection foundation.

3.2 Project 2: MQTT Integration

Learning Objectives:

- Integrate MQTT client with DroneKit telemetry
- Publish vessel data to MQTT broker
- Use environment variables for configuration
- Establish foundation for multi-vessel communication

What's New:

- MQTT client setup and authentication
- Publishing telemetry data to a topic
- Environment variable usage with python-dotenv
- Adaptive TLS configuration (secure/non-secure based on .env settings)

Project Structure:

Create the project directory:

```
# From the code directory
cd ~/lab04-companion/code
mkdir 2_proj
cd 2_proj
```

Create the enhanced script:

`leader_telemetry.py`

```python
from dronekit import connect
```

```python
import time
from datetime import datetime
import paho.mqtt.client as mqtt
from dotenv import load_dotenv
import os
import ssl

# Load environment variables from the .env file located one directory above
load_dotenv(os.path.join(os.path.dirname(__file__), '..', '.env'))

# MQTT connection details from environment variables
MQTT_BROKER = os.getenv('MQTT_BROKER')
MQTT_PORT = int(os.getenv('MQTT_PORT'))
MQTT_USERNAME = os.getenv('MQTT_USERNAME')
MQTT_PASSWORD = os.getenv('MQTT_PASSWORD')
MQTT_USE_TLS = os.getenv('MQTT_USE_TLS', 'false').lower() == 'true'
MQTT_TOPIC = 'leader/position'  # Hardcoded topic for this project

# Connect to the vehicle (SITL) using the specified connection string
connection_string = 'udp:127.0.0.1:14550'
print(f"Connecting to vehicle on: {connection_string}")

# Establish a connection to the vehicle
vehicle = connect(connection_string, wait_ready=True)

# Function to publish telemetry data to MQTT broker
def publish_to_mqtt(client, payload):
    # Publish the payload to the specified MQTT topic
    result = client.publish(MQTT_TOPIC, payload)

    # Check the result of the publish operation
    if result.rc == mqtt.MQTT_ERR_SUCCESS:
        print("Successfully published to MQTT.")
    else:
        print(f"Failed to publish to MQTT: {result.rc}")

# Function to output basic telemetry data
def get_telemetry():
    # Get the current timestamp in the format dd/mm/yyyy - HH:MM:SS
    timestamp = datetime.now().strftime("%d/%m/%Y - %H:%M:%S")

    # Get current heading (yaw) in degrees from the vehicle
    heading = vehicle.heading

    # Get ground speed in meters per second and round to two decimal places
    ground_speed = round(vehicle.groundspeed, 2)

    # Get latitude and longitude from the vehicle's GPS location
    lat = vehicle.location.global_frame.lat
    lon = vehicle.location.global_frame.lon

    # Create a payload string for MQTT
    payload = f"Timestamp: {timestamp}, Heading: {heading} degrees, Ground Speed: 
{ground_speed} m/s, Latitude: {lat}, Longitude: {lon}"

    # Print the telemetry data to the console
    print(payload)

    # Publish the telemetry data to the MQTT broker
    publish_to_mqtt(client, payload)

# Initialize MQTT Client
client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)  # Use VERSION2 to suppress deprecation 
warning
client.username_pw_set(MQTT_USERNAME, MQTT_PASSWORD)

# Configure TLS if enabled in environment variables
if MQTT_USE_TLS:
    # Create an SSLContext that loads system CAs (which include Let's Encrypt)
    ctx = ssl.create_default_context()
    client.tls_set_context(ctx)
    print("TLS enabled, using system default CA certificates")
else:
    print("TLS disabled, using non-secure connection")

print(f"Connecting to {MQTT_BROKER}:{MQTT_PORT} ...")
client.connect(MQTT_BROKER, MQTT_PORT, 60)
client.loop_start()  # Start the loop in a separate thread
```

```python
try:
    # Main loop to continuously read and publish telemetry data every 5 seconds
    while True:
        get_telemetry()  # Call the function to get and publish telemetry data
        time.sleep(5)    # Wait for 5 seconds before the next data output

except KeyboardInterrupt:
    # Catch keyboard interrupt (Ctrl+C) to exit the script gracefully
    print("Exiting script...")

finally:
    # Ensure the vehicle connection is closed and MQTT loop is stopped
    vehicle.close()
    client.loop_stop()
    client.disconnect()
    print("Vehicle connection closed and MQTT client disconnected.")
```

Understanding the New Components:

1. **Environment Variables**: Loaded from `.env` file one directory up (`../.env`)
2. **MQTT Client**: Created with VERSION2 API to avoid deprecation warnings
3. **Authentication**: Username/password from environment variables
4. **Publishing**: Telemetry formatted as comma-separated string
5. **Threading**: `client.loop_start()` handles MQTT communication in background
6. **TLS Configuration**: Conditionally applied based on MQTT_USE_TLS environment variable

Running the Project:

Terminal 1 - Start SITL (same as Project 1):

```
cd ~/sitl-test
sim_vehicle.py -v Rover -L syros \
--add-param-file=/home/<username>/custom-parms/boat.parm \
--out=udp:127.0.0.1:14550 --console --map
```

Terminal 2 - Run the enhanced script:

```
cd ~/lab04-companion/code
source ss_venv/bin/activate
cd 2_proj
python leader_telemetry.py
```

Expected Output (Terminal 2):

```
Connecting to vehicle on: udp:127.0.0.1:14550
TLS enabled, using system default CA certificates
Connecting to smartmove-local.syros.aegean.gr:8883 ...
Timestamp: 19/06/2025 - 21:38:29, Heading: 51 degrees, Ground Speed: 0.03 m/s, Latitude:
37.4393228, Longitude: 24.9456117
Successfully published to MQTT.
Timestamp: 19/06/2025 - 21:38:34, Heading: 51 degrees, Ground Speed: 0.03 m/s, Latitude:
37.4393228, Longitude: 24.9456117
Successfully published to MQTT.
Timestamp: 19/06/2025 - 21:38:39, Heading: 51 degrees, Ground Speed: 0.03 m/s, Latitude:
37.4393228, Longitude: 24.9456117
Successfully published to MQTT.
```

[Optional] Terminal 3 - Subscribe to verify MQTT messages:

```
# Using mosquitto_sub to verify messages are being published
mosquitto_sub -h smartmove-local.syros.aegean.gr -p 1883 \
  -u asv1 -P asv1 \
  -t leader/position -v
```

Expected Output (Terminal 3):

```
Timestamp: 19/06/2025 - 21:38:34, Heading: 51 degrees, Ground Speed:
0.03 m/s, Latitude: 37.4393228, Longitude: 24.9456117
```

Important Note - **Cross-Connection Message Reception**:

Notice that our script uses a secure (TLS-enabled) connection to the MQTT broker, while the command-line tool accesses the same broker insecurely. This works because messages published to the broker can be received by subscribers regardless of connection security type. A Python script using secure connection (port 8883) will publish to the same topic space as subscribers using non-secure connections (port 1883). This allows for flexible testing and debugging using command-line tools like mosquitto_sub on the non-secure port while maintaining secure connections in production code.

Testing Complete Flow:

1. Arm and move the vehicle in SITL (Terminal 1)
2. Observe telemetry updates in Terminal 2
3. Verify MQTT messages arriving in Terminal 3
4. Notice the 5-second update interval

Key Improvements from Project 1:

Feature	**Project 1**	**Project 2**
Configuration	Hardcoded	Environment variables
Data Distribution	Local only	MQTT broadcast
External Access	None	Any MQTT subscriber
Use Case	Single vehicle	Fleet monitoring
Security Config	Hardcoded (.env dependent)	Flexible (.env adaptive)

Common Issues and Solutions:

Issue	Solution
MQTT connection failed	Verify broker address and credentials in .env
No MQTT messages	Check firewall, ensure broker is accessible
Import error for paho	Verify paho-mqtt is installed in venv

Understanding the Data Flow:

SITL Vehicle → MAVLink → DroneKit → Python Script → MQTT Broker → Any Subscriber

This creates a bridge between the autopilot and external systems, enabling:

- Shore station monitoring
- Multi-vessel coordination
- Data logging and analysis
- Real-time fleet visualization

Message Format:

The current format is human-readable but verbose:

```
Timestamp: 23/06/2025 - 10:45:12, Heading: 180 degrees, Ground Speed: 0.02 m/s, Latitude:
37.438788, Longitude: 24.945544
```

Future projects will improve this with **JSON formatting** for easier parsing.

Team Exercises:

1. **Message Rate Analysis**: Monitor MQTT traffic - how does message frequency affect broker performance?
2. **Multiple Subscribers**: Have team members subscribe to the same topic - do all receive the messages?

Preparing for Project 3:

While functional, this code mixes concerns (vehicle control, MQTT communication, main logic). Project 3 will refactor into a cleaner, modular architecture that's easier to maintain and extend.

3.3 Project 3: Modular Architecture

Learning Objectives:

- Refactor code into object-oriented design
- Separate concerns (MQTT, vehicle control, main logic)
- Create reusable components
- Establish architecture for complex projects

What's New:

- Class-based design with MQTTHandler and VesselController
- Separation of responsibilities
- Cleaner code organization
- Username appended to MQTT messages

Project Structure:

Create the project directory with multiple files:

```
# From the code directory
cd ~/lab04-companion/code
mkdir 3_proj
cd 3_proj
```

File 1: MQTT Handler Class

`mqtt_handler.py`

```python
import paho.mqtt.client as mqtt
import os
import ssl
from dotenv import load_dotenv

# Load environment variables from the .env file located one directory above
load_dotenv(os.path.join(os.path.dirname(__file__), '..', '.env'))

class MQTTHandler:
    def __init__(self):
        # Initialize MQTT connection details from environment variables
        self.broker = os.getenv('MQTT_BROKER')
        self.port = int(os.getenv('MQTT_PORT'))
        self.username = os.getenv('MQTT_USERNAME')
        self.password = os.getenv('MQTT_PASSWORD')
        self.use_tls = os.getenv('MQTT_USE_TLS', 'false').lower() == 'true'
        self.topic = 'leader/position'

        # Initialize MQTT client and set up connection
        self.client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
        self.client.username_pw_set(self.username, self.password)

        # Configure TLS if enabled in environment variables
        if self.use_tls:
            # Create an SSLContext that loads system CAs (which include Let's Encrypt)
            ctx = ssl.create_default_context()
            self.client.tls_set_context(ctx)
            print("TLS enabled, using system default CA certificates")
        else:
            print("TLS disabled, using non-secure connection")

        # Attempt to connect to MQTT broker with error handling
        try:
            print(f"Connecting to {self.broker}:{self.port} ...")
            self.client.connect(self.broker, self.port, 60)
            self.client.loop_start()
            print("Successfully connected to MQTT broker")
        except Exception as e:
            print(f"Failed to connect to MQTT broker: {e}")
            raise

    # Function to publish a message to the MQTT broker
    def publish(self, payload):
        # Append the username to the payload
        payload_with_username = f"{payload}, USER: {self.username}"

        # Publish the modified payload to the MQTT broker
        result = self.client.publish(self.topic, payload_with_username)

        if result.rc == mqtt.MQTT_ERR_SUCCESS:
            print("Successfully published to MQTT.")
        else:
            print(f"Failed to publish to MQTT: {result.rc}")

        # Return the actual payload that was sent
        return payload_with_username

    # Function to disconnect the MQTT client
    def disconnect(self):
        self.client.loop_stop()
        self.client.disconnect()
```

File 2: Vessel Controller Class

`vessel_controller.py`

```python
from dronekit import connect
from datetime import datetime

class VesselController:
    def __init__(self, connection_string='udp:127.0.0.1:14550'):
        # Initialize the connection to the vehicle
        print(f"Connecting to vehicle on: {connection_string}")
        self.vehicle = connect(connection_string, wait_ready=True)

    # Function to retrieve telemetry data from the vehicle
    def get_telemetry(self):
        # Get the current timestamp in the format dd/mm/yyyy - HH:MM:SS
        timestamp = datetime.now().strftime("%d/%m/%Y - %H:%M:%S")

        # Get current heading (yaw) in degrees from the vehicle
        heading = self.vehicle.heading

        # Get ground speed in meters per second and round to two decimal places
        ground_speed = round(self.vehicle.groundspeed, 2)

        # Get latitude and longitude from the vehicle's GPS location
        lat = self.vehicle.location.global_frame.lat
        lon = self.vehicle.location.global_frame.lon

        # Create a payload string for MQTT
        payload = f"Timestamp: {timestamp}, Heading: {heading} degrees, Ground Speed: 
{ground_speed} m/s, Latitude: {lat}, Longitude: {lon}"

        return payload

    # Function to close the vehicle connection
    def close_connection(self):
        self.vehicle.close()
        print("Vehicle connection closed.")
```

File 3: Main Application

`main.py`

```python
import time
from mqtt_handler import MQTTHandler
from vessel_controller import VesselController

def main():
    # Initialize MQTT handler and vessel controller
    mqtt_handler = MQTTHandler()
    vessel_controller = VesselController()

    try:
        # Main loop to get telemetry data and publish it every 5 seconds
        while True:
            # Get telemetry data from the vessel
            telemetry_data = vessel_controller.get_telemetry()

            # Publish telemetry data to the MQTT broker and get the actual payload sent
            published_payload = mqtt_handler.publish(telemetry_data)
            print(published_payload)

            # Wait for 5 seconds before the next data output
            time.sleep(5)

    except KeyboardInterrupt:
        # Catch keyboard interrupt (Ctrl+C) to exit the script gracefully
        print("Exiting script...")

    finally:
        # Ensure both MQTT and vehicle connections are closed before exiting
        vessel_controller.close_connection()
        mqtt_handler.disconnect()
        print("All connections closed. Exiting.")

# Run the main function if this script is executed
if __name__ == "__main__":
    main()
```

Understanding the Modular Design:

1. **Separation of Concerns**:

   - `MQTTHandler:` All MQTT-related functionality
   - `VesselController:` All vehicle/DroneKit functionality
   - `main.py:` Application flow and coordination

2. **Benefits**:

   - **Reusability**: Classes can be imported into other projects
   - **Testability**: Each component can be tested independently
   - **Maintainability**: Changes to MQTT don't affect vehicle code
   - **Readability**: Clear structure and responsibilities

3. **New Feature**:
   - Username now appended to messages for identification
   - Useful when multiple vessels publish to same topic

Running the Project:

Same SITL setup as before, but now run the modular version:

```
cd ~/lab04-companion/code/3_proj
python main.py
```

Expected Output:

```
TLS enabled, using system default CA certificates
Connecting to smartmove-local.syros.aegean.gr:8883 ...
Successfully connected to MQTT broker
Connecting to vehicle on: udp:127.0.0.1:14550
Successfully published to MQTT.
Timestamp: 19/06/2025 - 23:35:43, Heading: 50 degrees, Ground Speed: 0.01 m/s, Latitude:
37.4393225, Longitude: 24.9456122, USER: asv1
Successfully published to MQTT.
Timestamp: 19/06/2025 - 23:35:48, Heading: 50 degrees, Ground Speed: 0.02 m/s, Latitude:
```

```
37.4393224, Longitude: 24.9456122, USER: asv1
Successfully published to MQTT.
Timestamp: 19/06/2025 - 23:35:53, Heading: 50 degrees, Ground Speed: 0.02 m/s, Latitude:
37.4393224, Longitude: 24.9456122, USER: asv1
```

MQTT Messages now include username:

leader/position Timestamp: 23/06/2025 - 11:15:32, Heading: 180 degrees, Ground Speed: 0.02 m/s, Latitude: 37.438788, Longitude: 24.945544, USER: asv1

Class Diagram:

```
┌───────────────────────┐      ┌────────────────────────┐
│   main.py         │     │  MQTTHandler     │
│                   │─────▶│  - broker          │
│  - main()         │     │ - port           │
│                   │     │ - username       │
└───────────────────────┘          │ - publish()      │
          │                   │ - disconnect()   │
          │                   └────────────────────────┘
          │
          │                   ┌────────────────────────┐
          └──────────────────────▶│  VesselController │
                              │ - vehicle        │
                              │ - get_telemetry()│
                              │ - close_conn()   │
                              └────────────────────────┘
```

Team Exercises:

1. **Code Analysis**: Compare lines of code between Project 2 and Project 3 - which is more complex?

2. **Extension Planning**: How would you add a new feature (e.g., logging) to this modular design?

Preparing for Project 4:

This modular architecture provides the foundation for:

- Role-based vehicle control (scout, team vessels)
- Dynamic configuration based on command-line arguments
- Multiple vehicle management
- Advanced MQTT topic routing

The clean separation of concerns makes it easy to extend functionality without modifying core components.

3.4 Project 4: Multi-Vehicle Support

Learning Objectives:

- Implement role-based vehicle control
- Use command-line arguments for configuration
- Dynamically load environment variables based on role
- Enable multi-vehicle simulation scenarios

What's New:

- Command-line argument parsing with `argparse`
- Role-specific environment variable loading
- Dynamic connection string and topic selection
- Support for scout and team vessels
- WSL2 **multi-vehicle SITL configuration** with Mission Planner integration

Project Structure:

Create the project directory:

```
# From the code directory
cd ~/lab04-companion/code
mkdir 4_proj
cd 4_proj
```

File 1: Enhanced MQTT Handler

`mqtt_handler.py`

```
import paho.mqtt.client as mqtt
import os
import ssl
from dotenv import load_dotenv

# Load environment variables from the .env file located one directory above
load_dotenv(os.path.join(os.path.dirname(__file__), '..', '.env'))

class MQTTHandler:
    def __init__(self, role):
        self.role = role.upper()  # Convert to uppercase for consistency

        # Initialize MQTT connection details from role-specific environment variables
        self.broker = os.getenv('MQTT_BROKER')
        self.port = int(os.getenv('MQTT_PORT'))
        self.username = os.getenv(f'{self.role}_MQTT_USERNAME')
        self.password = os.getenv(f'{self.role}_MQTT_PASSWORD')
        self.topic = os.getenv(f'{self.role}_POSITION_TOPIC')
        self.use_tls = os.getenv('MQTT_USE_TLS', 'false').lower() == 'true'

        # Validate all required configuration is present
        if not all([self.broker, self.port, self.username, self.password, self.topic]):
            raise ValueError(f"Missing required MQTT configuration for role:
{self.role}")

        # Initialize MQTT client and set up connection
        self.client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
        self.client.username_pw_set(self.username, self.password)
```

```python
        # Configure TLS if enabled in environment variables
        if self.use_tls:
            # Create an SSLContext that loads system CAs (which include Let's Encrypt)
            ctx = ssl.create_default_context()
            self.client.tls_set_context(ctx)
            print("TLS enabled, using system default CA certificates")
        else:
            print("TLS disabled, using non-secure connection")

        # Attempt to connect to MQTT broker with error handling
        try:
            print(f"Connecting to {self.broker}:{self.port} ...")
            self.client.connect(self.broker, self.port, 60)
            self.client.loop_start()
            print("Successfully connected to MQTT broker")
        except Exception as e:
            print(f"Failed to connect to MQTT broker: {e}")
            raise

    # Function to publish a message to the MQTT broker
    def publish(self, payload):
        # Append the username to the payload
        payload_with_username = f"{payload}, USER: {self.username}"

        # Publish the modified payload to the MQTT broker
        result = self.client.publish(self.topic, payload_with_username)

        if result.rc == mqtt.MQTT_ERR_SUCCESS:
            print(f"Successfully published to MQTT topic: {self.topic}")
        else:
            print(f"Failed to publish to MQTT: {result.rc}")

        # Return the actual payload that was sent
        return payload_with_username

    # Function to disconnect the MQTT client
    def disconnect(self):
        self.client.loop_stop()
        self.client.disconnect()
```

File 2: Enhanced Vessel Controller

`vessel_controller.py`

```
from dronekit import connect
from datetime import datetime
import os
from dotenv import load_dotenv

# Load environment variables
load_dotenv(os.path.join(os.path.dirname(__file__), '..', '.env'))

class VesselController:
    def __init__(self, role):
        self.role = role.upper()  # Convert role to uppercase for consistency

        # Get the connection string based on the role
        connection_string = self.get_connection_string()

        # Initialize the connection to the vehicle
        print(f"Connecting to vehicle on: {connection_string}")
        self.vehicle = connect(connection_string, wait_ready=True)

    def get_connection_string(self):
        connection_string = os.getenv(f'{self.role}_CONNECTION_STRING')
        if not connection_string:
            raise ValueError(f"Missing connection string for role: {self.role}")
        return connection_string

    # Function to retrieve telemetry data from the vehicle
    def get_telemetry(self):
        # Get the current timestamp in the format dd/mm/yyyy - HH:MM:SS
        timestamp = datetime.now().strftime("%d/%m/%Y - %H:%M:%S")

        # Get current heading (yaw) in degrees from the vehicle
        heading = self.vehicle.heading
        # Get ground speed in meters per second and round to two decimal places
        ground_speed = round(self.vehicle.groundspeed, 2)

        # Get latitude and longitude from the vehicle's GPS location
        lat = self.vehicle.location.global_frame.lat
        lon = self.vehicle.location.global_frame.lon

        # Create a payload string for MQTT
        payload = f"Timestamp: {timestamp}, Heading: {heading} degrees, Ground Speed:
{ground_speed} m/s, Latitude: {lat}, Longitude: {lon}"

        return payload

    # Function to close the vehicle connection
    def close_connection(self):
        self.vehicle.close()
        print("Vehicle connection closed.")
```

File 3: Main Application with Role Support

`main.py`

```
import time
import argparse
from mqtt_handler import MQTTHandler
from vessel_controller import VesselController

def main():
    # Set up command-line argument parsing
    parser = argparse.ArgumentParser(description='Multi-vessel telemetry system')
    parser.add_argument('role', choices=['scout', 'team1', 'team2', 'team3'],
                        help='Vessel role (scout, team1, team2, or team3)')

    # Parse command-line arguments
    args = parser.parse_args()
    role = args.role
```

```python
    print(f"Starting {role} vessel...")

    try:
        # Initialize MQTT handler and vessel controller with the specified role
        mqtt_handler = MQTTHandler(role)
        vessel_controller = VesselController(role)

        # Main loop to get telemetry data and publish it every 5 seconds
        while True:
            # Get telemetry data from the vessel
            telemetry_data = vessel_controller.get_telemetry()

            # Publish telemetry data to the MQTT broker and get the actual payload sent
            published_payload = mqtt_handler.publish(telemetry_data)
            print(published_payload)

            # Wait for 5 seconds before the next data output
            time.sleep(5)

    except KeyboardInterrupt:
        # Catch keyboard interrupt (Ctrl+C) to exit the script gracefully
        print("Exiting script...")

    except Exception as e:
        # Catch any other errors (missing environment variables, connection failures, etc.)
        print(f"Error: {e}")

    finally:
        # Ensure both MQTT and vehicle connections are closed before exiting
        try:
            vessel_controller.close_connection()
            mqtt_handler.disconnect()
        except:
            pass  # Ignore errors during cleanup
        print("All connections closed. Exiting.")

# Run the main function if this script is executed
if __name__ == "__main__":
    main()
```

3.4.1 Running Multiple Vehicles

3.4.1.1 Prerequisites

Find your Windows host IP from WSL2:

```
ip route show | grep -i default | awk '{ print $3}'
# Example output: 172.17.80.1
```

3.4.1.2 SITL Instance Setup

Terminal 1 - Start SITL for Scout:

```
cd ~/sitl-test
sim_vehicle.py -v Rover -L Syros \
--add-param-file=/home/thomas/custom-parms/boat.parm \
--out=udp:127.0.0.1:14550 --console --map
```

Terminal 2 - Start SITL for Team1:

```
cd ~/sitl-test2  # Different directory
sim_vehicle.py -v Rover -L Syros2 \
--add-param-file=/home/thomas/custom-parms/boat.parm \
--instance 1 --console --map --sysid=2 \
--out=udp:127.0.0.1:14560 --out=udp:172.17.80.1:14550
```

Critical Note: Replace `172.17.80.1` with your actual Windows host IP.

3.4.1.3 Understanding MAVProxy Network Outputs

Scout Instance Outputs:

```
MANUAL> output
2 outputs
0: 172.17.80.1:14550    # Auto-detected by MAVProxy (Mission Planner)
```

```
1: 127.0.0.1:14550      # Explicit output (Python scripts)
```

Team1 Instance Outputs:

```
MANUAL> output
3 outputs
0: 172.17.80.1:14560    # Auto-detected (wrong port for Mission Planner)
1: 127.0.0.1:14560      # Explicit output (Python scripts)
2: 172.17.80.1:14550    # Manual addition (Mission Planner)
```

Key Insight: MAVProxy automatically detects WSL2 and adds Windows host outputs, but Instance 2 gets the wrong port (14560) for Mission Planner, requiring manual correction.

3.4.1.4 Mission Planner Integration

Connect Mission Planner:

1. Open Mission Planner on Windows
2. Select UDP connection type
3. Connect to port 14550
4. Both vehicles appear with System IDs 1 and 2

Verification: Check vehicle dropdown in Mission Planner shows both Scout (ID=1) and Team1 (ID=2).

3.4.1.5 Python Script Execution

Terminal 3 - Run Scout script:

```
cd ~/lab04-companion/code/4_proj
python main.py scout
```

Terminal 4 - Run Team1 script:

```
cd ~/lab04-companion/code/4_proj
python main.py team1
```

3.4.1.6 Expected Outputs

Scout Output:

```
TLS enabled, using system default CA certificates
Connecting to smartmove-local.syros.aegean.gr:8883 ...
Successfully connected to MQTT broker
Connecting to vehicle on: udp:127.0.0.1:14550
Timestamp: 23/06/2025 - 11:45:12, Heading: 180 degrees, Ground Speed: 0.02 m/s, Latitude:
37.438788, Longitude: 24.945544, USER: scout
Successfully published to MQTT topic: scout/position
```

Team1 Output:

```
TLS enabled, using system default CA certificates
Connecting to smartmove-local.syros.aegean.gr:8883 ...
Successfully connected to MQTT broker
Connecting to vehicle on: udp:127.0.0.1:14560
Timestamp: 23/06/2025 - 11:45:15, Heading: 180 degrees, Ground Speed: 0.00 m/s, Latitude:
37.438788, Longitude: 24.945544, USER: team1
Successfully published to MQTT topic: team1/position
```

3.4.1.7 Monitoring All Vessels

MQTT Subscription:

```
mosquitto_sub -h smartmove-local.syros.aegean.gr -p 1883 \
  -u scout -P scout \
  -t scout/position -t team1/position -t team2/position -t team3/position -v
```

3.4.2 Key Architectural Changes

- **Dynamic Configuration:** No hardcoded values - everything from environment
- **Role-Based Topics:** Each vessel publishes to its own topic
- **Scalability:** Easy to add more vessels by adding environment variables
- **Command-Line Control:** Simple role selection at runtime
- **WSL2 Integration:** Automatic Windows host detection with manual override capability
- **Unified Visualization:** Mission Planner displays all vehicles simultaneously

3.4.3 Troubleshooting Guide

Issue	Solution
Missing environment variable	Check .env has all role-specific variables
Connection refused	Ensure each SITL instance uses different port
Wrong role name	Use exact names: scout, team1, team2, team3
Only one vehicle in Mission Planner	Verify Team1 has `--out=udp:<host_ip>:14550`
Can't find Windows host IP	Run `ip route show | grep -i default | awk '{ print $3}'`
System IDs conflict	Verify `--sysid=2` for Team1, check with `param show SYSID_THISMAV`

3.4.4 Network Architecture Summary

Connection Flow:

- Python scripts → Localhost ports (14550, 14560) → Individual SITL instances
- SITL instances → Windows host IP:14550 → Mission Planner

- SITL instances → MQTT broker → Inter-vessel communication

This architecture enables:

- Independent programmatic control of each vessel
- Unified ground control station visualization
- Real-time inter-vessel communication via MQTT

Team Exercises:

1. **Multi-Vehicle Coordination**: Run all 4 roles simultaneously - observe MQTT traffic patterns
2. **Role Validation**: What happens if you use an invalid role name? How could you improve error handling?

Preparing for Project 5:

Current limitations:

- Messages are still comma-separated strings (hard to parse)
- No inter-vessel communication
- No subscription to other vessels' data

Project 5 will address these with JSON formatting and subscription capabilities.

3.5 Project 5: JSON Communication

Learning Objectives:

- Implement JSON message formatting
- Add MQTT subscription capabilities
- Enable vessel-to-vessel data exchange
- Separate scout and team vessel scripts

What's New:

- JSON format for structured data exchange
- Separate scout.py and team.py scripts
- MQTT subscription to receive other vessels' data
- Message parsing and data extraction

Project Structure:

Create the project directory:

# From the code directory

```
cd ~/lab04-companion/code
mkdir 5_proj
cd 5_proj
```

File 1: Enhanced MQTT Handler with Subscription

`mqtt_handler.py`

```python
import paho.mqtt.client as mqtt
import os
from dotenv import load_dotenv
import json

# Load environment variables from the .env file located one directory above
load_dotenv(os.path.join(os.path.dirname(__file__), '..', '.env'))

class MQTTHandler:
    def __init__(self, role):
        self.role = role.upper()

        # Initialize MQTT connection details from role-specific environment variables
        self.broker = os.getenv('MQTT_BROKER')
        self.port = int(os.getenv('MQTT_PORT'))
        self.username = os.getenv(f'{self.role}_MQTT_USERNAME')
        self.password = os.getenv(f'{self.role}_MQTT_PASSWORD')
        self.topic = os.getenv(f'{self.role}_POSITION_TOPIC')

        if not all([self.broker, self.port, self.username, self.password, self.topic]):
            raise ValueError(f"Missing required MQTT configuration for role: {self.role}")

        # Initialize MQTT client and set up connection
        self.client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
        self.client.username_pw_set(self.username, self.password)
        self.client.connect(self.broker, self.port, 60)
        self.client.loop_start()

    def publish(self, payload):
        # Add the boat identifier to the payload
        payload['boat'] = self.username

        # Convert the payload to a JSON string
        json_payload = json.dumps(payload)

        # Publish the JSON payload to the MQTT broker
        result = self.client.publish(self.topic, json_payload)

        if result.rc == mqtt.MQTT_ERR_SUCCESS:
            print(f"Successfully published to MQTT topic: {self.topic}")
        else:
            print(f"Failed to publish to MQTT: {result.rc}")

    def subscribe(self, topic_name, callback=None):
        topic = os.getenv(topic_name)
        if not topic:
            raise ValueError(f"Missing {topic_name} in environment variables")

        def default_callback(client, userdata, msg):
            try:
                payload = json.loads(msg.payload.decode())
                print(f"Received message on topic {msg.topic}:")
                print(f"  Latitude: {payload.get('latitude')}, Longitude:
{payload.get('longitude')}")
            except json.JSONDecodeError:
                print("Received invalid JSON message")
            except KeyError:
                print("Received message doesn't contain expected data")

        self.client.subscribe(topic)
        self.client.message_callback_add(topic, callback or default_callback)

    def disconnect(self):
        self.client.loop_stop()
        self.client.disconnect()
```

File 2: Enhanced Vessel Controller with JSON Output

`vessel_controller.py`

```
from dronekit import connect
from datetime import datetime
import os
from dotenv import load_dotenv

# Load environment variables
load_dotenv(os.path.join(os.path.dirname(__file__), '..', '.env'))

class VesselController:
    def __init__(self, role):
        self.role = role.upper()  # Convert role to uppercase for consistency

        # Get the connection string based on the role
        connection_string = self.get_connection_string()

        # Initialize the connection to the vehicle
        print(f"Connecting to vehicle on: {connection_string}")
        self.vehicle = connect(connection_string, wait_ready=True)

    def get_connection_string(self):
        connection_string = os.getenv(f'{self.role}_CONNECTION_STRING')
        if not connection_string:
            raise ValueError(f"Missing connection string for role: {self.role}")
        return connection_string

    # Function to retrieve telemetry data from the vehicle
    def get_telemetry(self):
        # Get the current timestamp in the format dd/mm/yyyy - HH:MM:SS
        timestamp = datetime.now().strftime("%d/%m/%Y - %H:%M:%S")

        # Create a dictionary with telemetry data
        telemetry_data = {
            "timestamp": timestamp,
            "heading": self.vehicle.heading,
            "ground_speed": round(self.vehicle.groundspeed, 2),
            "latitude": self.vehicle.location.global_frame.lat,
            "longitude": self.vehicle.location.global_frame.lon
        }

        return telemetry_data

    # Function to close the vehicle connection
    def close_connection(self):
        self.vehicle.close()
        print("Vehicle connection closed.")
```

File 3: Scout Vessel Script

`scout.py`

```
import time
import os
from dotenv import load_dotenv
from mqtt_handler import MQTTHandler
from vessel_controller import VesselController

# Load environment variables from the .env file located one directory above
load_dotenv(os.path.join(os.path.dirname(__file__), '..', '.env'))
```

```python
def main():
    print("Starting scout vessel...")

    try:
        # Initialize MQTT handler and vessel controller for scout
        mqtt_handler = MQTTHandler('SCOUT')
        vessel_controller = VesselController('SCOUT')

        # Main loop to get telemetry data and publish it every 5 seconds
        while True:
            # Get telemetry data from the vessel
            telemetry_data = vessel_controller.get_telemetry()

            # Publish telemetry data to the MQTT broker and get the actual payload sent
            published_payload = mqtt_handler.publish(telemetry_data)
            print(published_payload)

            # Wait for 5 seconds before the next data output
            time.sleep(5)

    except KeyboardInterrupt:
        # Catch keyboard interrupt (Ctrl+C) to exit the script gracefully
        print("Exiting script...")

    except Exception as e:
        # Catch any other errors (missing environment variables, connection failures, etc.)
        print(f"Error: {e}")

    finally:
        # Ensure both MQTT and vehicle connections are closed before exiting
        try:
            vessel_controller.close_connection()
            mqtt_handler.disconnect()
        except:
            pass  # Ignore errors during cleanup
        print("All connections closed. Exiting.")

if __name__ == "__main__":
    main()
```

File 4: Team Vessel Script

`team.py`

```python
import time
import argparse
import os
from dotenv import load_dotenv
from mqtt_handler import MQTTHandler
from vessel_controller import VesselController

# Load environment variables from the .env file located one directory above
load_dotenv(os.path.join(os.path.dirname(__file__), '..', '.env'))

def main():
    # Set up command-line argument parsing
    parser = argparse.ArgumentParser(description='Team vessel telemetry system with scout
subscription')
    parser.add_argument('team', choices=['team1', 'team2', 'team3'],
                        help='Team vessel role (team1, team2, or team3)')

    # Parse command-line arguments
    args = parser.parse_args()
    team = args.team

    print(f"Starting {team} vessel...")

    try:
        # Initialize MQTT handler and vessel controller for the specified team
        mqtt_handler = MQTTHandler(team)
        vessel_controller = VesselController(team)

        # Subscribe to scout position topic
        mqtt_handler.subscribe('SCOUT_POSITION_TOPIC')

        # Main loop to get telemetry data and publish it every 5 seconds
        while True:
            # Get telemetry data from the vessel
            telemetry_data = vessel_controller.get_telemetry()
            # Publish telemetry data to the MQTT broker and get the actual payload sent
            published_payload = mqtt_handler.publish(telemetry_data)
            print(published_payload)

            # Wait for 5 seconds before the next data output
            time.sleep(5)

    except KeyboardInterrupt:
        # Catch keyboard interrupt (Ctrl+C) to exit the script gracefully
        print("Exiting script...")

    except Exception as e:
        # Catch any other errors (missing environment variables, connection failures, etc.)
        print(f"Error: {e}")

    finally:
        # Ensure both MQTT and vehicle connections are closed before exiting
        try:
            vessel_controller.close_connection()
            mqtt_handler.disconnect()
        except:
            pass  # Ignore errors during cleanup
        print("All connections closed. Exiting.")

if __name__ == "__main__":
    main()
```

3.5.1 Running JSON Communication System

3.5.1.1 Multi-Vehicle SITL Requirements

Essential Setup: Project 5 requires multiple independent SITL instances for vessel-to-vessel communication testing. Each vessel (scout, team1, team2, team3) needs its own SITL instance running on separate ports.

Minimum Configuration:

- **2 SITL instances**: Scout + Team1 (basic JSON communication)
- **3+ SITL instances**: Add Team2/Team3 for full multi-vessel scenarios

Quick SITL Setup Reminder:

```
# Terminal 1 - Scout SITL:
sim_vehicle.py -v Rover -L Syros --add-param-file=/home/thomas/custom-parms/boat.parm --map
--console --out=udp:127.0.0.1:14550
```

```
# Terminal 2 - Team1 SITL:
sim_vehicle.py -v Rover -L Syros2 --add-param-file=/home/thomas/custom-parms/boat.parm
--instance 1 --console --map --sysid=2 --out=udp:127.0.0.1:14560 --out=udp:172.17.80.1:14550
```

Critical: Each SITL instance must use different ports (14550, 14560, 14570, 14580) and system IDs. See **Project 4 Section 3.4.1.2** for complete multi-vehicle SITL configuration.

3.5.1.2 Prerequisites

Complete Project 4 SITL setup (Windows host IP discovery, MAVProxy configuration). See Section 3.4.1.1-3.4.1.2 for detailed multi-vehicle SITL configuration.

3.5.1.3 Script Execution

Terminal 3 - Run Scout (JSON Publisher):

```
cd ~/lab04-companion/code/5_proj
python scout.py
```

Terminal 4 - Run Team1 (JSON Publisher + Scout Subscriber):

```
cd ~/lab04-companion/code/5_proj
python team.py team1
```

Critical Note: Start scout first to ensure team vessels can immediately receive position data.

3.5.1.4 Expected JSON Outputs

Scout Output:

```
Starting scout vessel...
TLS enabled, using system default CA certificates
Connecting to smartmove-local.syros.aegean.gr:8883 ...
Successfully connected to MQTT broker
Connecting to vehicle on: udp:127.0.0.1:14550
Successfully published to MQTT topic: scout/position
{"timestamp": "20/06/2025 - 07:48:18", "heading": 180, "ground_speed": 0.02, "latitude":
37.4393258, "longitude": 24.9456158, "boat": "scout"}
```

Team1 Output (with Scout Subscription):

```
Starting team1 vessel...
TLS enabled, using system default CA certificates
Connecting to smartmove-local.syros.aegean.gr:8883 ...
Successfully connected to MQTT broker
Connecting to vehicle on: udp:127.0.0.1:14560
Successfully published to MQTT topic: team1/position
{"timestamp": "20/06/2025 - 07:49:01", "heading": 180, "ground_speed": 0.03, "latitude":
37.4391456, "longitude": 24.9453889, "boat": "team1"}
Received message on topic scout/position:
  Latitude: 37.4393258, Longitude: 24.9456158
```

3.5.1.5 Understanding JSON Message Flow

Message Structure Evolution:

- **Project 4**: String format with USER appended:

```
"Timestamp: 23/06/2025 - 11:45:12, Heading: 180 degrees, Ground
Speed: 0.02 m/s, Latitude: 37.438788, Longitude: 24.945544, USER:
scout"
```

- **Project 5**: Structured JSON with boat identifier

JSON Payload Components:

```
{
  "timestamp": "20/06/2025 - 07:48:18",    # Local system time
  "heading": 180,                          # Vehicle compass heading (degrees)
  "ground_speed": 0.02,                    # Speed in m/s (rounded to 2 decimals)
  "latitude": 37.4393258,                  # GPS latitude (decimal degrees)
  "longitude": 24.9456158,                 # GPS longitude (decimal degrees)
  "boat": "scout"                          # Role identifier (added by MQTT handler)
}
```

Inter-Vessel Communication Pattern:

1. Scout publishes to `scout/position` topic every 5 seconds
2. Team vessels subscribe to `scout/position` on startup
3. Team vessels receive and parse scout JSON messages
4. Team vessels publish their own telemetry every 5 seconds.

3.5.1.6 Monitoring JSON Communication

MQTT JSON Subscription:

```
mosquitto_sub -h smartmove-local.syros.aegean.gr -p 1883 \
  -u scout -P scout \
  -t scout/position -t team1/position -v
```

Expected JSON Stream:

```
scout/position {"timestamp": "20/06/2025 - 07:48:18", "heading": 180, "ground_speed": 0.02,
"latitude": 37.4393258, "longitude": 24.9456158, "boat": "scout"}
team1/position {"timestamp": "20/06/2025 - 07:49:01", "heading": 180, "ground_speed": 0.03,
"latitude": 37.4391456, "longitude": 24.9453889, "boat": "team1"}
```

3.5.2 Key Architectural Changes from Project 4

JSON Communication Layer: Structured data exchange replaces string concatenation
Subscription System: Team vessels actively listen for scout position updates
Script Separation: Dedicated scout.py and team.py for role-specific behavior
Message Parsing: Automatic JSON deserialization with error handling **Data Visualization**: Enhanced console output shows exact transmitted JSON

3.5.3 Troubleshooting Guide

Issue	Solution
Team vessel not receiving scout messages	Start scout.py first, ensure both use same MQTT broker
"Received invalid JSON message"	Check MQTT message format, verify JSON serialization
Missing boat identifier in JSON	Verify MQTT handler adds boat field before transmission
Subscription callback not triggered	Check topic environment variables, verify subscription setup
JSON parsing errors	Validate telemetry data dictionary format in VesselController
Mixed string/JSON format	Ensure all publish() calls return JSON strings, not dictionaries

3.5.4 JSON Communication Architecture Summary

Data Flow Evolution:

```
Project 4: Dict → String → MQTT → Console Display
Project 5: Dict → JSON → MQTT → JSON Parse → Structured Display
```

Communication Patterns:

- **Unidirectional**: Scout broadcasts position to all team vessels
- **Structured**: JSON format enables programmatic data extraction
- **Asynchronous**: Team vessels process scout data during main loop
- **Scalable**: Multiple team vessels can subscribe to same scout feed

Network Architecture:

```
Scout (14550) → JSON → scout/position topic
```

```
Team1 (14560) → JSON → team1/position topic
Team1 (14560) ← JSON ← scout/position topic (subscription)
```

This architecture enables structured inter-vessel data exchange while maintaining all Project 4 capabilities (TLS, multi-vehicle SITL, Mission Planner integration).

Team Exercises:

1. **Message Analysis**: Use `mosquitto_sub` with `-v` flag to see raw JSON messages
2. **Data Flow Visualization**: Draw a diagram showing how data flows from scout to team vessels

Preparing for Project 6:

The foundation is now set for:

- Command processing (follow/stop commands)
- More complex inter-vessel coordination
- Decision-making based on received data

3.6 Project 6: Command Processing

Learning Objectives:

- Implement bidirectional MQTT communication
- Process commands received via MQTT
- Handle both JSON and plaintext message formats
- Create foundation for autonomous behaviors

What's New:

- Command topic subscription for each team

- Flexible message parsing (JSON with plaintext fallback)
- Command handler with visual feedback
- Strategic design decision on message formats

Project Structure:

Create the project directory:

```
# From the code directory
cd ~/lab04-companion/code
mkdir 6_proj
cd 6_proj
```

File 1: MQTT Handler with Command Processing

`mqtt_handler.py`

```python
import paho.mqtt.client as mqtt
import os
import ssl
from dotenv import load_dotenv
import json

# Load environment variables from the .env file located one directory above
load_dotenv(os.path.join(os.path.dirname(__file__), '..', '.env'))

class MQTTHandler:
    def __init__(self, role):
        self.role = role.upper()  # Convert to uppercase for consistency

        # Initialize MQTT connection details from role-specific environment variables
        self.broker = os.getenv('MQTT_BROKER')
        self.port = int(os.getenv('MQTT_PORT'))
        self.username = os.getenv(f'{self.role}_MQTT_USERNAME')
        self.password = os.getenv(f'{self.role}_MQTT_PASSWORD')
        self.topic = os.getenv(f'{self.role}_POSITION_TOPIC')
        self.use_tls = os.getenv('MQTT_USE_TLS', 'false').lower() == 'true'
        # Validate all required configuration is present
        if not all([self.broker, self.port, self.username, self.password, self.topic]):
            raise ValueError(f"Missing required MQTT configuration for role: {self.role}")

        # Initialize MQTT client and set up connection
        self.client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
        self.client.username_pw_set(self.username, self.password)

        # Configure TLS if enabled in environment variables
        if self.use_tls:
            # Create an SSLContext that loads system CAs (which include Let's Encrypt)
            ctx = ssl.create_default_context()
            self.client.tls_set_context(ctx)
            print("TLS enabled, using system default CA certificates")
        else:
            print("TLS disabled, using non-secure connection")

        # Attempt to connect to MQTT broker with error handling
        try:
            print(f"Connecting to {self.broker}:{self.port} ...")
            self.client.connect(self.broker, self.port, 60)
            self.client.loop_start()
            print("Successfully connected to MQTT broker")
        except Exception as e:
            print(f"Failed to connect to MQTT broker: {e}")
            raise

    # Function to publish a message to the MQTT broker
    def publish(self, payload):
        # Add the boat identifier to the payload
        payload['boat'] = self.username

        # Convert the payload to a JSON string
        json_payload = json.dumps(payload)

        # Publish the JSON payload to the MQTT broker
        result = self.client.publish(self.topic, json_payload)
        if result.rc == mqtt.MQTT_ERR_SUCCESS:
            print(f"Successfully published to MQTT topic: {self.topic}")
        else:
            print(f"Failed to publish to MQTT: {result.rc}")

        # Return the actual payload that was sent
        return json_payload

    def subscribe(self, topic_names):
        # Handle single topic or list of topics
        if isinstance(topic_names, str):
            topic_names = [topic_names]

        for topic_name in topic_names:
            topic = os.getenv(topic_name)
            if not topic:
                raise ValueError(f"Missing {topic_name} in environment variables")

            self.client.subscribe(topic)
            self.client.message_callback_add(topic, self.on_message)
            print(f"Subscribed to topic: {topic}")

    def on_message(self, client, userdata, msg):
        # First, try to parse the message as JSON
        try:
            payload = json.loads(msg.payload.decode())
            print(f"Received message on topic {msg.topic}:")

            if 'latitude' in payload and 'longitude' in payload:
                print(f"  Latitude: {payload['latitude']}, Longitude: {payload['longitude']}")
            elif msg.topic.lower().endswith('commands'):
                command = payload.get('command', '').lower()
                self.handle_command(command)
            else:
                print(f"  Payload: {payload}")

        except json.JSONDecodeError:
            # If JSON parsing fails, treat it as plain text command if from command topic
            if msg.topic.lower().endswith('commands'):
                command = msg.payload.decode().strip().lower()
                self.handle_command(command)
            else:
                print(f"Received invalid JSON message on topic {msg.topic}:
{msg.payload.decode()}")

    def handle_command(self, command):
        if command == "follow":
            self.print_command_message("Command to start following is issued")
        elif command == "stop":
            self.print_command_message("Command to stop following is issued")
        else:
            self.print_command_message(f"Unknown command received: {command}")

    def print_command_message(self, message):
        print("\n" + "*" * 50)
        print(message)
        print("*" * 50 + "\n")

    # Function to disconnect the MQTT client
    def disconnect(self):
        self.client.loop_stop()
        self.client.disconnect()
```

File 2: Updated Team Script

`team.py`

```
import time
import argparse
import os
from dotenv import load_dotenv
from mqtt_handler import MQTTHandler
from vessel_controller import VesselController

# Load environment variables from the .env file located one directory above
load_dotenv(os.path.join(os.path.dirname(__file__), '..', '.env'))
```

```python
def main():
    # Set up command-line argument parsing
    parser = argparse.ArgumentParser(description='Team vessel with command processing and scout 
subscription')
    parser.add_argument('team', choices=['team1', 'team2', 'team3'],
                        help='Team vessel role (team1, team2, or team3)')

    # Parse command-line arguments
    args = parser.parse_args()
    team = args.team

    print(f"Starting {team} vessel...")

    try:
        # Initialize MQTT handler and vessel controller for the specified team
        mqtt_handler = MQTTHandler(team)
        vessel_controller = VesselController(team)

        # Create a list of topics to subscribe to
        topics_to_subscribe = ['SCOUT_POSITION_TOPIC', f'{team.upper()}_COMMANDS']

        # Subscribe to topics
        mqtt_handler.subscribe(topics_to_subscribe)

        # Main loop to get telemetry data and publish it every 5 seconds
        while True:
            # Get telemetry data from the vessel
            telemetry_data = vessel_controller.get_telemetry()

            # Publish telemetry data to the MQTT broker and get the actual payload sent
            published_payload = mqtt_handler.publish(telemetry_data)
            print(published_payload)

            # Wait for 5 seconds before the next data output
            time.sleep(5)

    except KeyboardInterrupt:
        # Catch keyboard interrupt (Ctrl+C) to exit the script gracefully
        print("Exiting script...")

    except Exception as e:
        # Catch any other errors (missing environment variables, connection failures, etc.)
        print(f"Error: {e}")

    finally:
        # Ensure both MQTT and vehicle connections are closed before exiting
        try:
            vessel_controller.close_connection()
            mqtt_handler.disconnect()
        except:
            pass  # Ignore errors during cleanup
        print("All connections closed. Exiting.")

if __name__ == "__main__":
    main()
```

Important Note: scout.py and vessel_controller.py remain the same as Project 5.

3.6.1 Running Command Processing System

3.6.1.1 Multi-Vehicle SITL Requirements

Same multi-vehicle SITL setup as Project 5.
Minimum 2 SITL instances required for command testing:
Scout (14550) + Team1 (14560).
See Project 4 Section 3.4.1.2 for complete SITL configuration.

3.6.1.2 Script Execution

Terminal 3 - Run Scout (Position Publisher):

```
cd ~/lab04-companion/code/6_proj
```

```
python scout.py
```

Terminal 4 - Run Team1 (Position Publisher + Command Receiver):

```
cd ~/lab04-companion/code/6_proj
python team.py team1
```

Terminal 5 - Command Center (New):

```
# Send follow command to team1
mosquitto_pub -h smartmove-local.syros.aegean.gr -p 1883 \
  -u team1 -P team1 \
  -t team1/commands -m "follow"
```

```
# Send stop command to team1
mosquitto_pub -h smartmove-local.syros.aegean.gr -p 1883 \
  -u team1 -P team1 \
  -t team1/commands -m "stop"
```

3.6.1.3 Expected Command Processing Outputs

Scout Output (Unchanged from Project 5):

```
Starting scout vessel...
TLS enabled, using system default CA certificates
Successfully connected to MQTT broker
{"timestamp": "20/06/2025 - 08:19:45", "heading": 180, "ground_speed": 0.02, "latitude":
37.4377008, "longitude": 24.9437918, "boat": "scout"}
```

Team1 Output (Enhanced with Command Processing):

```
Starting team1 vessel...
TLS enabled, using system default CA certificates
```

```
Successfully connected to MQTT broker
Subscribed to topic: scout/position
Subscribed to topic: team1/commands
{"timestamp": "20/06/2025 - 08:19:50", "heading": 180, "ground_speed": 0.03, "latitude":
37.4391457, "longitude": 24.9453888, "boat": "team1"}
Received message on topic scout/position:
  Latitude: 37.4377008, Longitude: 24.9437918

**************************************************
Command to start following is issued
**************************************************

Successfully published to MQTT topic: team1/position
{"timestamp": "20/06/2025 - 08:19:55", "heading": 180, "ground_speed": 0.03, "latitude":
37.4391456, "longitude": 24.9453888, "boat": "team1"}
```

3.6.1.4 Understanding Command & Control Architecture

Communication Evolution:

- **Project 5**: Scout → Position Data → Teams (unidirectional telemetry)
- **Project 6**: Scout → Position Data → Teams + Command Center → Commands → Individual Teams (bidirectional control)

Command Processing Flow:

1. **Command Center** sends targeted commands to specific team vessels
2. **Team Vessels** subscribe to both scout position and their individual command topic
3. **Message Parser** handles both JSON telemetry and plaintext commands
4. **Command Handler** processes commands and provides visual feedback
5. **Infrastructure Ready** for autonomous behavior implementation (Project 7)

Flexible Message Format Strategy:

```
Position Messages: JSON format for structured data
Command Messages: Plaintext format for operator simplicity
```

Multi-Topic Subscription Pattern:

```
topics_to_subscribe = ['SCOUT_POSITION_TOPIC', f'{team.upper()}_COMMANDS']
# team1 subscribes to: scout/position + team1/commands
# team2 subscribes to: scout/position + team2/commands
```

3.6.1.5 Command Testing Scenarios

Individual Team Control:

```
# Command only team1 to follow
mosquitto_pub -h smartmove-local.syros.aegean.gr -p 1883 \
  -u team1 -P team1 -t team1/commands -m "follow"
```

```
# Command only team2 to stop (if running)
mosquitto_pub -h smartmove-local.syros.aegean.gr -p 1883 \
  -u team2 -P team2 -t team2/commands -m "stop"
```

Command Verification: Each team vessel displays received commands with distinctive visual borders, confirming successful command reception and processing.

3.6.1.6 Monitoring Complete System

MQTT Traffic Monitoring:

```
# Monitor all position data and commands
mosquitto_sub -h smartmove-local.syros.aegean.gr -p 1883 \
  -u scout -P scout \
  -t scout/position -t team1/position -t team1/commands -v
```

Expected Multi-Channel Traffic:

```
scout/position {"timestamp": "20/06/2025 - 08:19:45", "boat": "scout", ...}
team1/position {"timestamp": "20/06/2025 - 08:19:50", "boat": "team1", ...}
team1/commands follow
team1/commands stop
```

3.6.2 Key Architectural Changes from Project 5

- **Bidirectional Communication**: Teams now receive commands in addition to publishing telemetry
- **Command Infrastructure**: Foundation for autonomous behavior control without implementation
- **Multi-Topic Architecture**: Each team monitors both scout position and individual command channels
- **Message Format Flexibility**: JSON for telemetry, plaintext for commands (operator-friendly design)
- **Targeted Control**: Individual team command channels enable independent vessel coordination
- **Visual Command Feedback**: Distinctive asterisk-bordered messages confirm command reception

3.6.3 Troubleshooting Guide

Issue	**Solution**
Commands not received by team vessel	Verify correct team name in topic (team1/commands), check credentials match team role
No command visual feedback	Ensure command topic ends with 'commands', verify message reaches vessel
Command processed but no action	Expected behavior - movement implementation in Project 7
Multiple teams receive same command	Verify using correct team-specific topic (team1/commands vs team2/commands)
JSON parsing errors on commands	Expected behavior - commands use plaintext fallback parsing

Command topic subscription failed	Check environment variables contain TEAM1_COMMANDS, TEAM2_COMMANDS definitions

3.6.4 Command & Control Architecture Summary

Operational Scenario:

```
Naval Convoy with Remote Command Capability
│
├── Scout Vessel: Autonomous navigation + position broadcasting
├── Team Vessels: Autonomous capability + command reception + scout tracking
└── Command Center: Remote command authority over individual vessels
```

Communication Patterns:

- **Telemetry Flow**: All vessels → MQTT broker → Monitoring systems
- **Command Flow**: Command center → Specific team vessel (targeted control)
- **Position Sharing**: Scout → All team vessels (situational awareness)

Strategic Design Decision: Project 6 establishes **command infrastructure without (yet) autonomous behavior implementation**. This separation enables:

- Reliable command reception testing
- Communication layer validation
- **Foundation for complex autonomous behaviors** (Project 7)
- Educational progression from communication to control

Network Architecture:

```
Scout (14550) → JSON → scout/position topic
Team1 (14560) → JSON → team1/position topic
Team1 (14560) ← Commands ← team1/commands topic
Command Center → Plaintext → team1/commands topic
```

This architecture creates a hybrid autonomy model where vessels operate independently but receive high-level behavioral commands from human operators, establishing the foundation for coordinated multi-vessel autonomous operations.

Team Exercises:

1. **Command Testing**: Send various commands (valid and invalid) - observe handling
2. **Message Format Comparison**: Which format would you choose for production systems? Why?

Preparing for Project 7:

Currently, commands are received and displayed but not acted upon. Project 7 will:

- Implement actual vehicle control based on commands
- Create follow-the-leader behavior
- Handle distance-based decision making
- Demonstrate complete autonomous coordination
- Implement proper MQTT QoS levels (telemetry = 0, commands = 1)

3.7 Project 7: Follow-the-Leader Implementation

Learning Objectives:

- Implement autonomous follow behavior
- Control vehicle modes and movement via DroneKit
- Handle GPS inaccuracies with smart logic
- Create state-based vehicle control system

What's New:

- Vehicle arming and mode control
- Distance calculations between vessels
- Follow/hold/stop state management
- GPS movement detection with averaging

- Autonomous navigation based on scout position

Project Structure:

Create the project directory:

```
# From the code directory
cd ~/lab04-companion/code
mkdir 7_proj
cd 7_proj
```

File 1: Enhanced Vessel Controller with Follow Logic

`vessel_controller.py`

```python
from dronekit import connect, VehicleMode, LocationGlobalRelative
from datetime import datetime
import os
from dotenv import load_dotenv
from math import radians, sin, cos, sqrt, atan2
import time
from collections import deque

# Load environment variables
load_dotenv(os.path.join(os.path.dirname(__file__), '..', '.env'))

class VesselController:
    def __init__(self, role):
        self.role = role.upper()  # Convert role to uppercase for consistency
        self.following = False
        self.last_scout_distance = None
        self.last_goto_time = time.time()
        self.last_goto_position = None
        self.last_report_time = 0
        self.report_interval = 3  # Report every 3 seconds
        self.scout_speeds = deque(maxlen=3)  # Store last 3 speed readings

        # Get the connection string based on the role
        connection_string = self.get_connection_string()

        # Initialize the connection to the vehicle
        print(f"Connecting to vehicle on: {connection_string}")
        self.vehicle = connect(connection_string, wait_ready=True)

    def get_connection_string(self):
        connection_string = os.getenv(f'{self.role}_CONNECTION_STRING')
        if not connection_string:
            raise ValueError(f"Missing connection string for role: {self.role}")
        return connection_string

    # Function to retrieve telemetry data from the vehicle
    def get_telemetry(self):
        # Get the current timestamp in the format dd/mm/yyyy - HH:MM:SS
        timestamp = datetime.now().strftime("%d/%m/%Y - %H:%M:%S")

        # Create a dictionary with telemetry data
        telemetry_data = {
            "timestamp": timestamp,
            "heading": self.vehicle.heading,
            "ground_speed": round(self.vehicle.groundspeed, 2),
            "latitude": self.vehicle.location.global_frame.lat,
            "longitude": self.vehicle.location.global_frame.lon
        }

        return telemetry_data

    def arm_vehicle(self):
        # Arm the vehicle if it's not already armed
        if not self.vehicle.armed:
            print("Arming vehicle...")
            self.vehicle.armed = True
            while not self.vehicle.armed:
                time.sleep(1)
            print("Vehicle armed.")

    def calculate_distance(self, lat1, lon1, lat2, lon2):
        # Calculate the great circle distance between two points on earth
        R = 6371000  # Earth radius in meters
        lat1, lon1, lat2, lon2 = map(radians, [lat1, lon1, lat2, lon2])

        dlat = lat2 - lat1
        dlon = lon2 - lon1

        a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
        c = 2 * atan2(sqrt(a), sqrt(1-a))

        return R * c

    def report_status(self, distance):
        current_time = time.time()
        if current_time - self.last_report_time >= self.report_interval:
            mode = "LOITERING" if self.following and distance < 5 else \
                   "STOPPED" if not self.following else "FOLLOWING"
            print(f"Distance to scout: {distance:.2f} meters. Mode: {mode}")
            self.last_report_time = current_time

    def scout_has_moved(self, scout_lat, scout_lon, scout_speed):
        if self.last_goto_position is None:
            return True

        distance_moved = self.calculate_distance(
            self.last_goto_position[0], self.last_goto_position[1],
            scout_lat, scout_lon
        )

        self.scout_speeds.append(scout_speed)
        avg_speed = sum(self.scout_speeds) / len(self.scout_speeds)

        # Consider scout moved if position changed >4m OR average speed >0.5 m/s
        return distance_moved > 4 or avg_speed > 0.5

    def follow_scout(self, scout_lat, scout_lon, scout_speed):
        current_time = time.time()

        # Calculate distance to scout
        current_distance = self.calculate_distance(
            self.vehicle.location.global_frame.lat,
            self.vehicle.location.global_frame.lon,
            scout_lat, scout_lon
        )

        self.report_status(current_distance)

        if self.following:
            # Check if too close to scout
            if current_distance < 5:
                if self.vehicle.mode.name != "LOITER":
                    self.vehicle.mode = VehicleMode("LOITER")
                    print("Too close to scout. Loitering to maintain position.")
            else:
                # Resume following if needed
                if self.vehicle.mode.name != "GUIDED":
                    self.vehicle.mode = VehicleMode("GUIDED")
                    print("Resuming follow mode.")

                # Determine if we should issue a new goto command
                should_update = (
                    self.last_goto_position is None or
                    current_time - self.last_goto_time >= 15 or
                    current_distance > self.last_scout_distance + 5
                )

                if should_update and self.scout_has_moved(scout_lat, scout_lon, scout_speed):
                    print(f"Issuing new goto command. Distance to scout:
{current_distance:.2f} meters")
                    self.vehicle.simple_goto(LocationGlobalRelative(scout_lat, scout_lon, 0))
                    self.last_goto_time = current_time
                    self.last_goto_position = (scout_lat, scout_lon)
                    self.last_scout_distance = current_distance

    def set_guided_mode(self):
        # Set the vehicle mode to GUIDED and initialize following state
        self.vehicle.mode = VehicleMode("GUIDED")
        while self.vehicle.mode.name != "GUIDED":
            time.sleep(1)
         print("Vehicle is in GUIDED mode. Started following.")
         self.following = True

     def stop_following(self):
         # Stop following by switching to LOITER mode
         self.vehicle.mode = VehicleMode("LOITER")
         while self.vehicle.mode.name != "LOITER":
             time.sleep(1)
         print("Vehicle is in LOITER mode. Stopped following.")
         self.following = False
         self.last_scout_distance = None
         self.last_goto_time = None
         self.last_goto_position = None
         self.scout_speeds.clear()

     # Function to close the vehicle connection
     def close_connection(self):
         self.vehicle.close()
         print("Vehicle connection closed.")
```

File 2: Enhanced Team Script with Command Actions

`team.py`

```
import time
import argparse
import os
from dotenv import load_dotenv
from mqtt_handler import MQTTHandler
from vessel_controller import VesselController
import json
import struct

# Load environment variables from the .env file located one directory above
load_dotenv(os.path.join(os.path.dirname(__file__), '..', '.env'))

def on_message(client, userdata, msg):
    vessel_controller = userdata['vessel_controller']
```

```python
    try:
        payload = json.loads(msg.payload.decode())

        # Handle scout position updates
        if 'latitude' in payload and 'longitude' in payload and 'ground_speed' in payload:
            if vessel_controller.following:
                vessel_controller.follow_scout(
                    payload['latitude'],
                    payload['longitude'],
                    payload['ground_speed']
                )
        # Handle commands
        elif msg.topic.lower().endswith('commands'):
            command = payload.get('command', '').lower()
            handle_command(command, vessel_controller)

    except json.JSONDecodeError:
        # Handle plaintext commands
        if msg.topic.lower().endswith('commands'):
            command = msg.payload.decode().strip().lower()
            handle_command(command, vessel_controller)
        else:
            print(f"Received invalid message on topic {msg.topic}: {msg.payload.decode()}")

    except Exception as e:
        print(f"Error processing message: {e}")

def handle_command(command, vessel_controller):
    if command == "follow":
        print("\n" + "*" * 50)
        print("Command to start following is issued")
        print("*" * 50 + "\n")
        vessel_controller.arm_vehicle()
        vessel_controller.set_guided_mode()

    elif command == "stop":
        print("\n" + "*" * 50)
        print("Command to stop following is issued")
        print("*" * 50 + "\n")
        vessel_controller.stop_following()
    else:
        print(f"Unknown command received: {command}")

def main():
    # Set up command-line argument parsing
    parser = argparse.ArgumentParser(description='Team vessel with autonomous follow behavior')
    parser.add_argument('team', choices=['team1', 'team2', 'team3'],
                       help='Team vessel role (team1, team2, or team3)')

    # Parse command-line arguments
    args = parser.parse_args()
    team = args.team

    print(f"Starting {team} vessel...")

    try:
        # Initialize MQTT handler and vessel controller for the specified team
        vessel_controller = VesselController(team)
        mqtt_handler = MQTTHandler(team)

        # Create a list of topics to subscribe to
        topics_to_subscribe = ['SCOUT_POSITION_TOPIC', f'{team.upper()}_COMMANDS']

        # Set vessel controller in userdata for callback access
        mqtt_handler.client.user_data_set({'vessel_controller': vessel_controller})

        # Subscribe to topics with custom callback
        # QoS will be automatically set to 1 for commands, 0 for positions
        mqtt_handler.subscribe(topics_to_subscribe, on_message)

        print("Team vessel ready. Waiting for commands...")

        # Main loop to get telemetry data and publish it every 5 seconds
        # Note: MQTT messages are processed automatically by the background thread
        while True:
            try:
                # Get telemetry data from the vessel
```

```python
            telemetry_data = vessel_controller.get_telemetry()

            # Publish telemetry data to the MQTT broker with QoS 0
            published_payload = mqtt_handler.publish(telemetry_data, qos=0)
            print(published_payload)

        except struct.error:
            print("Encountered a malformed MQTT message. Attempting to reconnect...")
            mqtt_handler.client.disconnect()
            time.sleep(5)
            try:
                mqtt_handler.client.reconnect()
                mqtt_handler.subscribe(topics_to_subscribe, on_message)
            except Exception as e:
                print(f"Reconnection failed: {e}")
                time.sleep(5)  # Wait before retrying

        except Exception as e:
            print(f"An error occurred in the main loop: {e}")

        # Wait for 5 seconds before the next telemetry output
        time.sleep(5)

except KeyboardInterrupt:
    # Catch keyboard interrupt (Ctrl+C) to exit the script gracefully
    print("\nShutting down team vessel...")

except Exception as e:
    # Catch any other errors (missing environment variables, connection failures, etc.)
    print(f"Error: {e}")

finally:
    # Ensure both MQTT and vehicle connections are closed before exiting
    try:
        vessel_controller.close_connection()
        mqtt_handler.disconnect()
    except:
        pass  # Ignore errors during cleanup
    print("All connections closed. Exiting.")
```

```python
if __name__ == "__main__":
    main()
```

File 3: Enhanced Scout Script with Command Actions

`scout.py`

```python
import time
import os
from dotenv import load_dotenv
from mqtt_handler import MQTTHandler
from vessel_controller import VesselController

# Load environment variables from the .env file located one directory above
load_dotenv(os.path.join(os.path.dirname(__file__), '..', '.env'))

def main():
    print("Starting scout vessel...")

    try:
        # Initialize MQTT handler and vessel controller for scout
        mqtt_handler = MQTTHandler('SCOUT')
        vessel_controller = VesselController('SCOUT')

        print("Scout vessel ready. Publishing telemetry...")

        # Main loop to get telemetry data and publish it every 5 seconds
        # Note: MQTT connection is maintained automatically by the background thread
        while True:
            try:
                # Get telemetry data from the vessel
                telemetry_data = vessel_controller.get_telemetry()

                # Publish telemetry data to the MQTT broker with QoS 0
                # Scout positions are frequent updates, QoS 0 is appropriate
                published_payload = mqtt_handler.publish(telemetry_data, qos=0)
                print(published_payload)
            except Exception as e:
                print(f"An error occurred in the main loop: {e}")
                # Try to reconnect on error
                try:
                    mqtt_handler.client.reconnect()
                except:
                    print("Reconnection failed, will retry next cycle")
                    time.sleep(5)

            # Wait for 5 seconds before the next data output
            time.sleep(5)

    except KeyboardInterrupt:
        # Catch keyboard interrupt (Ctrl+C) to exit the script gracefully
        print("\nShutting down scout vessel...")

    except Exception as e:
        # Catch any other errors (missing environment variables, connection failures, etc.)
        print(f"Error: {e}")

    finally:
        # Ensure both MQTT and vehicle connections are closed before exiting
        try:
            vessel_controller.close_connection()
            mqtt_handler.disconnect()
        except:
            pass  # Ignore errors during cleanup
        print("All connections closed. Exiting.")

if __name__ == "__main__":
    main()
```

File 4: Enhanced MQTT Handler Script with QoS Implementation

`mqtt_handler.py`

```python
import paho.mqtt.client as mqtt
import os
import ssl
from dotenv import load_dotenv
import json

# Load environment variables from the .env file located one directory above
load_dotenv(os.path.join(os.path.dirname(__file__), '..', '.env'))

class MQTTHandler:
    def __init__(self, role):
        self.role = role.upper()  # Convert to uppercase for consistency

        # Initialize MQTT connection details from role-specific environment variables
        self.broker = os.getenv('MQTT_BROKER')
        self.port = int(os.getenv('MQTT_PORT'))
        self.username = os.getenv(f'{self.role}_MQTT_USERNAME')
        self.password = os.getenv(f'{self.role}_MQTT_PASSWORD')
        self.topic = os.getenv(f'{self.role}_POSITION_TOPIC')
        self.use_tls = os.getenv('MQTT_USE_TLS', 'false').lower() == 'true'

        # Validate all required configuration is present
        if not all([self.broker, self.port, self.username, self.password, self.topic]):
            raise ValueError(f"Missing required MQTT configuration for role: {self.role}")

        # Initialize MQTT client and set up connection
        self.client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
        self.client.username_pw_set(self.username, self.password)

        # Configure TLS if enabled in environment variables
        if self.use_tls:
            # Create an SSLContext that loads system CAs (which include Let's Encrypt)
            ctx = ssl.create_default_context()
            self.client.tls_set_context(ctx)
            print("TLS enabled, using system default CA certificates")
        else:
            print("TLS disabled, using non-secure connection")

        # Attempt to connect to MQTT broker with error handling
        try:
            print(f"Connecting to {self.broker}:{self.port} ...")
            self.client.connect(self.broker, self.port, 60)
            # Start background thread for MQTT event processing
            self.client.loop_start()
            print("Successfully connected to MQTT broker")
        except Exception as e:
            print(f"Failed to connect to MQTT broker: {e}")
            raise

    # Function to publish a message to the MQTT broker
    def publish(self, payload, qos=0):
        # Add the boat identifier to the payload
        payload['boat'] = self.username

        # Convert the payload to a JSON string
        json_payload = json.dumps(payload)

        # Publish the JSON payload to the MQTT broker with specified QoS
        result = self.client.publish(self.topic, json_payload, qos=qos)

        if result.rc == mqtt.MQTT_ERR_SUCCESS:
            print(f"Successfully published to MQTT topic: {self.topic} (QoS={qos})")
        else:
            print(f"Failed to publish to MQTT: {result.rc}")

        # Return the actual payload that was sent
        return json_payload

    def subscribe(self, topic_names, callback=None, qos=None):
        # Handle single topic or list of topics
        if isinstance(topic_names, str):
            topic_names = [topic_names]

        for i, topic_name in enumerate(topic_names):
            topic = os.getenv(topic_name)
            if not topic:
                raise ValueError(f"Missing {topic_name} in environment variables")
            # Determine QoS for this topic
            if qos is None:
                # Default QoS based on topic type
                if 'commands' in topic_name.lower():
                    topic_qos = 1  # QoS 1 for commands (guaranteed delivery)
                else:
                    topic_qos = 0  # QoS 0 for telemetry/positions
            elif isinstance(qos, list):
                topic_qos = qos[i]
            else:
                topic_qos = qos

            self.client.subscribe(topic, qos=topic_qos)

            # Use custom callback if provided, otherwise use built-in callback
            if callback:
                self.client.message_callback_add(topic, callback)
            else:
                self.client.message_callback_add(topic, self.on_message)

            print(f"Subscribed to topic: {topic} (QoS={topic_qos})")

    def on_message(self, client, userdata, msg):
        # Built-in callback for backward compatibility (Project 6 style)
        try:
            payload = json.loads(msg.payload.decode())
            print(f"Received message on topic {msg.topic}:")

            if 'latitude' in payload and 'longitude' in payload:
                print(f"  Latitude: {payload['latitude']}, Longitude: {payload['longitude']}")
            elif msg.topic.lower().endswith('commands'):
                command = payload.get('command', '').lower()
                self.handle_command(command)
            else:
                print(f"  Payload: {payload}")

        except json.JSONDecodeError:
            # If JSON parsing fails, treat it as plain text command if from command topic
            if msg.topic.lower().endswith('commands'):
                command = msg.payload.decode().strip().lower()
                self.handle_command(command)
            else:
                print(f"Received invalid JSON message on topic {msg.topic}:
{msg.payload.decode()}")

    def handle_command(self, command):
        if command == "follow":
            self.print_command_message("Command to start following is issued")
        elif command == "stop":
            self.print_command_message("Command to stop following is issued")
        else:
            self.print_command_message(f"Unknown command received: {command}")

    def print_command_message(self, message):
        print("\n" + "*" * 50)
        print(message)
        print("*" * 50 + "\n")

    # Function to disconnect the MQTT client
    def disconnect(self):
        self.client.loop_stop()  # Stop the background thread
        self.client.disconnect()
        print("MQTT connection closed.")
```

3.7.1 Running Autonomous Follow System

3.7.1.1 Multi-Vehicle SITL Requirements

Same multi-vehicle SITL setup as Projects 5-6. Minimum 2 SITL instances required for autonomous testing: Scout (14550) + Team1 (14560). See previous projects for complete SITL configuration.

3.7.1.2 Script Execution

Terminal 3 - Run Scout (Autonomous Leader):

```
cd ~/lab04-companion/code/7_proj
python scout.py
```

Terminal 4 - Run Team1 (Autonomous Follower):

```
cd ~/lab04-companion/code/7_proj
python team.py team1
```

Terminal 5 - Command Center (Autonomous Control):

```
# Initiate autonomous following
mosquitto_pub -h smartmove-local.syros.aegean.gr -p 1883 \
  -u team1 -P team1 \
  -t team1/commands -m "follow"

# Stop autonomous following
mosquitto_pub -h smartmove-local.syros.aegean.gr -p 1883 \
  -u team1 -P team1 \
  -t team1/commands -m "stop"
```

3.7.1.3 Expected Autonomous Behavior Outputs

Team1 Output (Autonomous Following Active):

```
Starting team1 vessel...
TLS enabled, using system default CA certificates
Successfully connected to MQTT broker
Subscribed to topic: scout/position (QoS=0)
Subscribed to topic: team1/commands (QoS=1)
Team vessel ready. Waiting for commands...

**************************************************
Command to start following is issued
**************************************************
Arming vehicle...
Vehicle armed.
Vehicle is in GUIDED mode. Started following.
```

```
Distance to scout: 219.63 meters. Mode: FOLLOWING
Issuing new goto command. Distance to scout: 219.63 meters
Distance to scout: 187.45 meters. Mode: FOLLOWING
Issuing new goto command. Distance to scout: 187.45 meters
Distance to scout: 12.8 meters. Mode: FOLLOWING
Too close to scout. Loitering to maintain position.
Distance to scout: 4.2 meters. Mode: LOITERING
```

3.7.1.4 Understanding Autonomous Follow Logic

State-Based Autonomous Behavior:

- **STOPPED**: Vehicle in LOITER mode, not following scout
- **FOLLOWING**: Vehicle in GUIDED mode, actively navigating toward scout
- **LOITERING**: Vehicle in LOITER mode, actively maintaining position near scout (< 5 meters)

Maritime-Specific Mode Selection: LOITER mode is used instead of HOLD for maritime vessels because:

- **HOLD Mode**: Simply stops thrusters, allowing vessel to drift with currents/waves
- **LOITER Mode**: Actively maintains GPS position using thrusters to counteract environmental forces

Smart Navigation Decision Making:

```
# Movement triggers (any condition met):
# 1. Scout position changed > 4 meters
# 2. Scout average speed > 0.5 m/s over last 3 readings
# 3. Distance to scout increased > 5 meters since last command
# 4. More than 15 seconds since last goto command
```

Autonomous Vehicle Control Flow:

1. **Command Reception**: "follow" command triggers vehicle arming and GUIDED mode

2. **Distance Monitoring**: Continuous calculation of distance to scout using GPS coordinates
3. **Movement Logic**: Issues `simple_goto()` commands when scout moves significantly
4. **Collision Avoidance**: Automatically switches to HOLD mode when within 5 meters
5. **State Reporting**: Regular status updates every 3 seconds

3.7.1.5 QoS Message Quality Implementation

Quality of Service Differentiation:

```
Commands (QoS 1): Guaranteed delivery with acknowledgment
Telemetry (QoS 0): Best effort delivery, allows message loss
```

QoS Assignment Logic:

- **Command Topics** (`team1/commands`): QoS 1 for reliable autonomous control
- **Position Topics** (`scout/position`): QoS 0 for high-frequency telemetry
- **Automatic Detection**: MQTTHandler automatically assigns QoS based on topic name

Benefits:

- **Critical Commands**: "follow"/"stop" commands guaranteed to reach destination
- **Telemetry Efficiency**: Position updates can be dropped without affecting autonomous behavior
- **Network Optimization**: Reduces broker overhead for non-critical high-frequency data

3.7.1.6 Testing Autonomous Scenarios

Scenario 1: Basic Follow Behavior

```
# 1. Send follow command
mosquitto_pub -h smartmove-local.syros.aegean.gr -p 1883 \
  -u team1 -P team1 -t team1/commands -m "follow"

# 2. Move scout in MAVProxy
arm throttle
```

```
mode guided
guided 37.440 24.946 # Or (better) by right click on map and 'fly to...'

# Expected: Team1 automatically navigates toward new scout position
```

Scenario 2: (Crude and basic, just for proof-of-concept) Collision Avoidance

```
# Position scout and team1 close together, send follow command
# Expected: Team1 switches to HOLDING mode when within 5 meters
```

Scenario 3: Stop Autonomous Operation

```
mosquitto_pub -h smartmove-local.syros.aegean.gr -p 1883 \
  -u team1 -P team1 -t team1/commands -m "stop"

# Expected: Team1 switches to LOITER mode, stops following
```

3.7.1.7 Monitoring Autonomous Operations

MQTT QoS Traffic Monitoring:

```
mosquitto_sub -h smartmove-local.syros.aegean.gr -p 1883 \
  -u scout -P scout \
  -t scout/position -t team1/position -t team1/commands -v
```

Expected Multi-QoS Traffic:

```
scout/position {"timestamp": "20/06/2025 - 12:42:31", "boat": "scout", ...}
team1/position {"timestamp": "20/06/2025 - 12:42:35", "boat": "team1", ...}
team1/commands follow
```

3.7.2 Key Architectural Changes from Project 6

- **Autonomous Vehicle Control**: Commands now trigger actual DroneKit vehicle movement and mode changes
- **QoS Message Differentiation**: Critical commands use guaranteed delivery, telemetry uses best effort

- **State-Based Follow Logic**: Sophisticated distance calculations and movement decision making
- **GPS Accuracy Handling**: Movement detection with speed averaging to handle GPS inaccuracies
- **Collision Avoidance**: Automatic proximity detection with hold behavior

3.7.3 Troubleshooting Guide

Issue	Solution
Vehicle doesn't arm on follow command	Check SITL instance is running, verify vehicle connection string
"Too close to scout" shows but vessel drifts	Normal behavior - LOITER mode actively maintains position against currents
No autonomous movement after follow	Check vehicle mode shows GUIDED, verify scout position is being received
Commands not received reliably	QoS 1 should ensure delivery - check MQTT broker connection
Distance calculations incorrect	Verify GPS coordinates are valid, check vehicle.location.global_frame
Vehicle stops following randomly	Check for GPS signal loss, verify SITL stability
Vessel drifting when stopped	Expected with LOITER - vessel actively fights drift to maintain GPS position

3.7.4 Autonomous Follow Architecture Summary

Operational Scenario:

```
Autonomous Naval Convoy with Intelligent Following
├── Scout Vessel: Independent navigation + position broadcasting (QoS 0)
├── Team Vessels: Autonomous follow behavior + command reception (QoS 1)
└── Command Center: High-level behavioral commands with guaranteed delivery
```

Autonomous Decision Making:

- **Movement Intelligence**: Smart goto command timing based on scout movement patterns
- **Safety Systems**: Automatic collision avoidance with proximity-based hold behavior
- **GPS Handling**: Robust movement detection accounting for GPS accuracy limitations
- **State Management**: Clear STOPPED/FOLLOWING/HOLDING states with visual feedback

Message Quality Architecture:

```
High-Frequency Telemetry (QoS 0) → Best effort delivery
Critical Commands (QoS 1) → Guaranteed delivery with broker acknowledgment
Background Thread Processing → Event-driven message handling
```

Real-World Applications:

- **Search and Rescue**: Autonomous vessel formation following lead search vessel
- **Convoy Operations**: Unmanned supply vessels following manned lead ship
- **Patrol Missions**: Coordinated autonomous patrol with dynamic formation control
- **Research Expeditions**: Autonomous data collection vessels following research platform

This architecture creates a fully autonomous multi-vessel system where vehicles make intelligent navigation decisions while maintaining reliable command and control communication, establishing the foundation for complex autonomous maritime operations.

Team Exercises:

1. **Parameter Tuning**: Experiment with different follow distances and update intervals
2. **Multi-follower Test**: Have multiple teams follow the same scout - observe behavior

Summary:

Project 7 demonstrates a complete autonomous maritime system where:

- Vessels respond to high-level commands
- Navigation adapts to environmental conditions
- Multiple vessels can coordinate via MQTT

This forms the foundation for real-world autonomous maritime operations.

3.8 Project 8: Mission Management (Conceptual)

Learning Objectives:

- Understand mission file formats and parsing
- Explore DroneKit mission upload capabilities
- Monitor mission execution progress
- Integrate mission status with MQTT reporting

What's New:

- File-based mission definition
- Programmatic mission upload
- AUTO mode execution
- Progress tracking and reporting

Note: This project provides **conceptual guidance rather than complete code**, encouraging teams to research DroneKit's mission management documentation for exact implementation details.

Mission File Format:

Create a sample mission file:

```
# syros_patrol.txt
# Maritime patrol pattern around Syros port
# Format: latitude,longitude,altitude_relative_to_home, …
```

Conceptual Implementation Structure:

```python
# mission_manager.py (conceptual outline)
from dronekit import connect, VehicleMode
import time
from mqtt_handler import MQTTHandler  # Reuse from previous projects

class MissionManager:
    def __init__(self, vehicle, mqtt_handler):
        self.vehicle = vehicle
        self.mqtt_handler = mqtt_handler
        self.mission_waypoints = []

    def load_mission_from_file(self, filename):
        """
        Parse mission file and create waypoint list

        Steps to implement:
        1. Open and read the file
        2. Parse each line (skip comments starting with #)
        3. Extract lat, lon, alt values
        4. Store in mission_waypoints list
        """
        waypoints = []
        # Implementation needed:
        # - Read file line by line
        # - Parse coordinates
        # - Handle errors gracefully
        return waypoints

    def upload_mission_to_vehicle(self):
        """
        Upload waypoints to vehicle using DroneKit commands

        Research needed:
        - How to clear existing mission
        - How to create mission items/commands
        - How to upload to vehicle
        - How to verify upload success

        Key DroneKit concepts to explore:
        - vehicle.commands
        - Command objects
        - MAV_CMD_NAV_WAYPOINT
        """
        print("Uploading mission to vehicle...")
        # Implementation needed:
        # - Clear current mission
        # - Create command sequence
        # - Upload to vehicle
        # - Verify upload

    def start_mission(self):
        """
        Arm vehicle and set AUTO mode
        """
        # Arm vehicle if not armed
        if not self.vehicle.armed:
            print("Arming vehicle...")
            self.vehicle.armed = True
            # Wait for arming

        # Set AUTO mode
        print("Setting AUTO mode...")
        self.vehicle.mode = VehicleMode("AUTO")
        # Wait for mode change
```

```python
    def monitor_mission_progress(self):
        """
        Track mission execution and publish updates

        Key parameters to monitor:
        - Current waypoint index
        - Total waypoints
        - Distance to next waypoint
        - Mission completion status

        MQTT updates could include:
        - Progress percentage
        - Current waypoint
        - ETA to completion
        """
        while self.vehicle.mode.name == "AUTO":
            # Get current mission status
            # Calculate progress
            # Publish via MQTT

            mission_update = {
                "type": "mission_progress",
                "current_wp": "?",  # Research how to get this
                "total_wp": len(self.mission_waypoints),
                "progress_percent": "?",
                "mode": self.vehicle.mode.name
            }

            self.mqtt_handler.publish(mission_update)
            time.sleep(5)
```

Possible Integration Approach:

```python
# main_mission.py (conceptual)
def main():
    # Connect to vehicle (reuse connection logic)
    vehicle = connect('udp:127.0.0.1:14550', wait_ready=True)

    # Initialize MQTT (reuse from previous projects)
    mqtt_handler = MQTTHandler('SCOUT')

    # Create mission manager
    mission_mgr = MissionManager(vehicle, mqtt_handler)

    # Load mission
    mission_mgr.load_mission_from_file('syros_patrol.txt')

    # Upload to vehicle
    mission_mgr.upload_mission_to_vehicle()

    # Start mission
    mission_mgr.start_mission()

    # Monitor progress
    mission_mgr.monitor_mission_progress()
```

Key Research Areas:

- **DroneKit Mission Commands**
    - Explore `vehicle.commands` documentation
    - Understand `CommandSequence` objects
    - Learn about `MAV_CMD_NAV_WAYPOINT` parameters

Mission Upload Process

Typical pattern (verify in documentation):

1. Clear existing: vehicle.commands.clear()

2. Add waypoints: vehicle.commands.add(cmd)

3. Upload: vehicle.commands.upload()

4. Monitoring: Progress Tracking

- How to access current waypoint index
- How to calculate distance to waypoint
- How to detect mission completion

Testing Approach:

1. **Start with Simple Mission**: 2-3 waypoints only
2. **Verify Upload**: Check mission in MAVProxy with `wp_list`
3. **Monitor Execution**: Watch map display during AUTO mode
4. **Debug with Logs**: Print status at each step

Common Challenges:

Challenge	**Research Direction**
Mission format	MAVLink mission item structure
Coordinate frames	Global vs relative altitude
Command types	NAV_WAYPOINT vs other commands
Upload verification	Checking command acknowledgments

MQTT Mission Updates:

Extend the MQTT system to broadcast mission events:

```
{
    "type": "mission_loaded",
    "waypoint_count": 6,
    "estimated_distance": 520.5
}

{
    "type": "mission_progress",
    "current_waypoint": 3,
    "total_waypoints": 6,
    "progress_percent": 50
}

{
    "type": "mission_complete",
    "total_time": 180,
    "final_position": {"lat": 37.438788, "lon": 24.945544}
}
```

Team Exercises:

1. **Documentation Research**: Find official DroneKit mission examples and adapt them
2. **Mission Planning**: Create different mission patterns (survey grid, perimeter patrol)
3. **Error Handling**: What happens if upload fails? How to detect and recover?

Integration with Previous Projects:

This mission management system could be combined with:

- **Project 6**: Receive mission files via MQTT commands
- **Project 7**: Switch between follow mode and mission mode
- **Future**: Dynamic mission modification based on discoveries

Summary:

Mission management adds another layer of autonomy, allowing vessels to:

- Execute pre-planned routes without constant supervision
- Report progress to shore stations
- Switch between different operational modes
- Build foundation for adaptive mission planning

Teams should research DroneKit documentation thoroughly before implementation, as mission command syntax and parameters require precise usage.

4 - Possible Advanced Challenges

From SITL Simulation to Real-World Implementation

These projects can (and probably should) start in SITL for algorithm development and testing, with several suitable for eventual real-world implementation.

Advanced Autonomous Behaviors

Dynamic Formation Control - Multiple vessels maintaining geometric formations while navigating to waypoints. Key concepts: relative positioning, formation mathematics, distributed coordination.

Adaptive Patrol Patterns - Vessels automatically adjusting patrol routes based on environmental conditions or mission requirements. Key concepts: dynamic path planning, environmental response.

Search and Rescue Coordination - Systematic area coverage with multiple vessels, including spiral searches and grid patterns. Key concepts: area decomposition, search optimization, coordination protocols.

Safety and Emergency Systems

Geofencing and Safe Return - Implementing virtual boundaries with automatic return-to-base when violated. Key concepts: boundary detection, emergency protocols, fail-safe behaviors.

Collision Avoidance with Maritime Rules - Following COLREGS (maritime right-of-way rules) for vessel encounters. Key concepts: maritime navigation rules, decision trees, priority-based maneuvering.

Emergency Communication Protocols - Handling communication failures and implementing backup control methods. Key concepts: redundancy, degraded operations, emergency beacons.

Computer Vision and Object Detection

Buoy and Marker Recognition - Detecting and navigating around navigation buoys, markers, or obstacles using camera feeds. Key concepts: computer vision, object classification, obstacle avoidance.

Autonomous Docking - Visual recognition of dock structures and precision maneuvering for automated berthing. Key concepts: visual servoing, precision control, depth perception.

Marine Wildlife Detection - Identifying and avoiding marine animals or debris in the water. Key concepts: real-time image processing, environmental awareness, protective behaviors.

Target Following and Inspection - Visually tracking and maintaining distance from floating objects or other vessels. Key concepts: object tracking, visual feedback control, relative motion estimation.

Emergency Response and Mission Override - Detecting critical situations (distress signals, oil spills, emergency flares) that trigger immediate mission bypassing to respond to higher-priority events. Key concepts: priority-based decision making, mission interruption protocols, emergency response hierarchies, autonomous mission replanning.

AIS Integration and Maritime Traffic

Commercial Traffic Avoidance - Using AIS receiver data to detect nearby commercial vessels and implement safe avoidance maneuvers. Key concepts: AIS data parsing, traffic prediction, safety corridors.

Harbor Approach Coordination - Monitoring AIS traffic to time harbor entries and coordinate with larger vessel movements. Key concepts: traffic analysis, timing optimization, maritime protocols.

Vessel Inspection Operations - Safely approaching AIS-tracked vessels for simulated inspection or assistance scenarios. Key concepts: approach protocols, relative positioning, safety distances.

Real-World Deployment Challenges

Power Management and Endurance - Monitoring battery levels and optimizing mission duration through speed/route adjustments. Key concepts: energy optimization, mission planning, resource constraints.

Weather-Adaptive Navigation - Adjusting behavior based on simulated wind, waves, or weather data. Key concepts: environmental sensing, adaptive control, safety margins.

Limited Connectivity Operations - Operating with intermittent MQTT connections and local decision-making. Key concepts: offline autonomy, data buffering, reconnection strategies.

Data Collection and Analysis

Environmental Monitoring Station - Vessels collecting and sharing environmental data while maintaining optimal sampling patterns. Key concepts: scientific data collection, spatial optimization, sensor integration.

Maritime Traffic Analysis - Monitoring and reporting on vessel movements in a simulated shipping lane. Key concepts: traffic pattern recognition, data logging, reporting systems.

Human-Machine Interaction

Voice Command Integration - Adding speech recognition for natural language vessel commands. Key concepts: natural language processing, command interpretation, user interfaces.

Real-Time Mission Updates - Dynamic mission modification through web interface or mobile app. Key concepts: mission flexibility, real-time planning, user interface design.

Fleet Coordination Scenarios

Harbor Operations Simulation - Coordinating multiple vessels in confined spaces with traffic management. Key concepts: traffic control, priority systems, spatial constraints.

Cooperative Towing Operations - Multiple vessels working together to move a simulated disabled vessel. Key concepts: cooperative control, force coordination, synchronized movement.

5 - Transitioning from SITL to Real-World Deployment

Physical Hardware Integration

When moving from simulation to actual hardware, the connection paradigm shifts from network-based communication (`udp:127.0.0.1:14550`) to direct serial communication with the autopilot. The Raspberry Pi 5 connects to the Pixhawk 2.4.8 via USB cable, creating a serial interface that typically appears as `/dev/ttyACM0` or `/dev/ttyUSB0` depending on the specific hardware configuration.

This transition requires updating connection strings to use direct serial communication (e.g., `/dev/ttyACM0`) and introduces the critical challenge of device enumeration consistency.

Persistent Device Naming with udev Rules

The Problem: Linux assigns device names based on detection order, meaning your autopilot might appear as `/dev/ttyACM0` when connected to one USB port but `/dev/ttyACM1` when connected to another, or even change between reboots. This creates unreliable connection strings that **could break your code**.

The Solution: udev rules create persistent, descriptive device names based on hardware characteristics rather than detection order. Instead of `/dev/ttyACM0`, your autopilot becomes `/dev/pixhawk1` regardless of which USB port is used.

Implementation Process:

- Identify the autopilot's unique serial number and vendor/product IDs using `lsusb` and `udevadm info`
- Create a custom udev rule that maps these hardware identifiers to a consistent device name
- Test the rule by disconnecting/reconnecting the device and verifying the persistent name appears
- The rule ensures `/dev/pixhawk1` always points to your specific autopilot hardware

Connection String Adaptations

Hardware deployment requires environment-specific configuration management. Your `.env` file expands to include hardware-specific connection strings alongside existing SITL parameters:

```
# SITL Configuration
SCOUT_CONNECTION_STRING=udp:127.0.0.1:14550

# Hardware Configuration
SCOUT_REAL_CONNECTION_STRING=/dev/pixhawk1
```

Your Python code implements conditional logic to select appropriate connection methods based on deployment mode.

Raspberry Pi Deployment Considerations

Software Installation: Transferring your development environment to the Raspberry Pi, including all Python dependencies and ensuring ARM64 compatibility for specialized libraries.

Auto-Start Integration: Implementing systemd services or startup scripts to automatically launch your vessel control software on boot, ensuring autonomous operation without manual intervention.

Network Configuration: Setting up 4G dongles or WiFi connections for remote MQTT communication, with considerations for data usage and connection reliability in maritime environments.

Power Management: Understanding power consumption patterns and implementing graceful shutdown procedures to prevent SD card corruption during power loss.

Common Deployment Challenges

Connection Verification: Systematic testing approaches to confirm Pixhawk connectivity before launching autonomous behaviors, including MAVLink heartbeat verification.

Hardware-Specific Debugging: Troubleshooting techniques for serial communication issues, including baud rate mismatches and cable connection problems.

Field Transition Strategy: Progressive testing methodology from bench setup to water deployment, ensuring each hardware integration step functions correctly before advancing to the next complexity level.

The **persistent device naming approach** becomes essential for reliable field operations, where manual intervention to fix connection string issues is often impractical or impossible. A lot of detailed information can be found online for this procedure, with a good example being this tutorial.