

# **FLYTBASE ASSIGNMENT**

*ASSIGNMENT REPORT*

*by*

**VARUN AJITH**

**29 MAR 2025**

# TABLE OF CONTENTS

<b>LIST OF FIGURES</b>	<b>ii</b>
<b>ABBREVIATIONS</b>	<b>iii</b>
<b>NOTATIONS</b>	<b>iv</b>
<b>1 INTRODUCTION</b>	<b>1</b>
1.1 Project Overview . . . . .	1
1.2 Problem Statement . . . . .	1
1.3 Scope and Objectives . . . . .	1
<b>2 SYSTEM ARCHITECTURE</b>	<b>3</b>
2.1 Overall Design . . . . .	3
2.2 Component Diagram . . . . .	4
2.3 Message Flow and Data Exchange . . . . .	4
<b>3 CORE IMPLEMENTATION</b>	<b>6</b>
3.1 Flight Schedule Node . . . . .	6
3.2 Deconfliction Service Node . . . . .	6
3.3 Spatial Conflict Checker Node . . . . .	7
3.4 Temporal Conflict Checker Node . . . . .	8
3.5 Visualization Node . . . . .	8
<b>4 ANALYSIS OF LIMITATIONS AND ERROR INVESTIGATION</b>	<b>10</b>
4.1 Memory Corruption and Waypoints Format Error . . . . .	10
4.2 Other Constraints and Unimplemented Features . . . . .	11
<b>5 CONCLUSION AND FUTURE WORK</b>	<b>13</b>
5.1 Key Achievements . . . . .	13
5.2 Lesson learned . . . . .	13
5.3 Future Enhancements . . . . .	13

## LIST OF FIGURES

2.1	Workflow . . . . .	4
3.1	Key Code Excerpt . . . . .	6
3.2	Orchestration Example . . . . .	7
3.3	Key Code Expert . . . . .	7
3.4	Overlap Check Example . . . . .	8
3.5	Visualization Code Snippet . . . . .	9
4.1	Error observed . . . . .	10
4.2	validation function . . . . .	11

## ABBREVIATIONS

<b>UAV</b>	Unmanned Aerial Vehicle
<b>ROS2</b>	Robot Operating System 2
<b>JSON</b>	JavaScript Object Notation
<b>ID</b>	Identity Document
<b>4D</b>	4-Dimensional
<b>GIL</b>	Global Interpreter Lock

## NOTATIONS

$m$       Meter,  $m$

# INTRODUCTION

## 1.1 Project Overview

The UAV Strategic Deconfliction System addresses a critical need in the rapidly expanding drone industry: preventing mid-air collisions between autonomous aerial vehicles operating in shared airspace. As commercial and recreational drone usage continues to grow exponentially, the risk of unintended encounters between these unmanned systems increases proportionally, necessitating robust conflict detection and resolution mechanisms.

## 1.2 Problem Statement

The core challenge is to design and implement a strategic deconfliction system that serves as the final authority for verifying whether a drone's planned waypoint mission is safe to execute within shared airspace. The system must perform comprehensive checks for conflicts in both space and time against the simulated flight paths of multiple other drones operating in the same region.

## 1.3 Scope and Objectives

The primary objectives of this implementation include:

- **Development Window:** The entire project was implemented within a 48-hour sprint, leading to some compromises in optimization and feature completeness.
- **Platform:** The system is built using ROS 2 Jazzy and Python, relying on libraries like Shapely for geometric computations.

- **Data:** Simulated drone trajectories are provided via a JSON file, representing flight schedules with waypoints  $(x, y, z)$  and associated time windows.

## SYSTEM ARCHITECTURE

### 2.1 Overall Design

The deconfliction system is designed as a distributed, service-oriented architecture. Its components communicate via ROS 2 services to enable parallel processing and modular code organization. The key nodes are:

- **Mission Input Node:**  
Receives primary mission details (waypoints and overall mission time window).
- **Deconfliction Service Node:**  
Acts as the orchestrator, querying the flight schedule database and invoking the spatial and temporal conflict checkers.
- **Spatial Conflict Checker Node:**  
Uses geometric algorithms to determine if the mission path comes too close to any drone trajectory.
- **Temporal Conflict Checker Node:**  
Verifies that the mission's time window does not conflict with the scheduled operations of other UAVs.
- **Flight Schedule Node:**  
Maintains a repository of simulated drone trajectories loaded from JSON.
- **Visualization Node:**  
Generates static plots and animations to visually demonstrate both safe and conflicting mission scenarios.



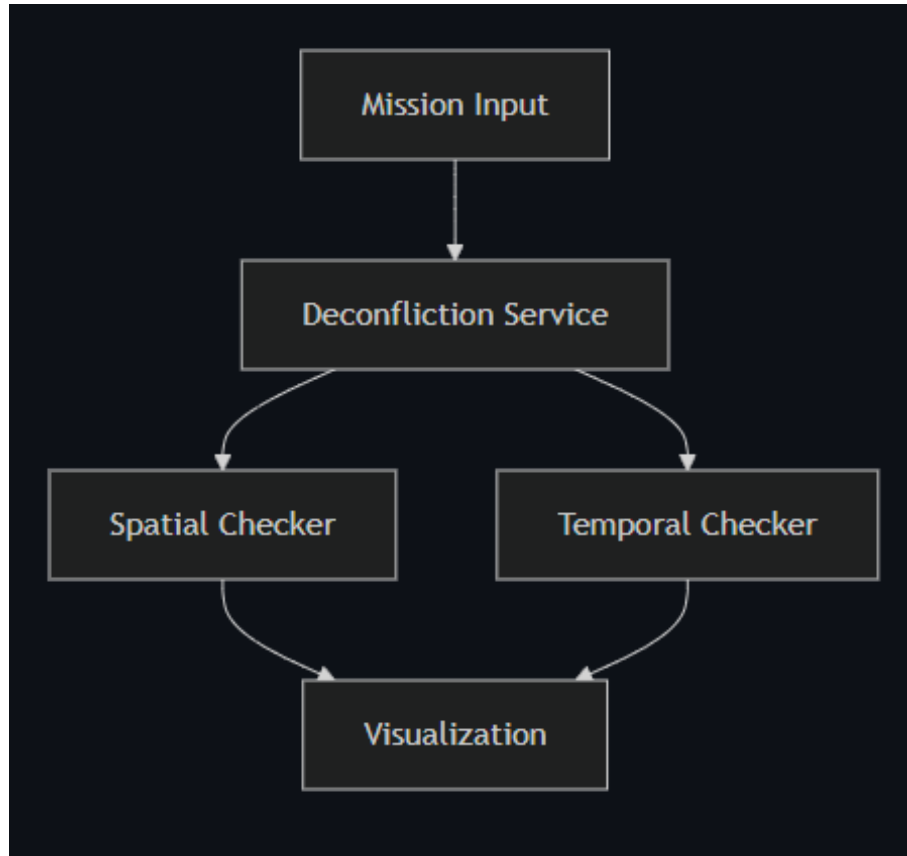


Figure 2.1: Workflow

## 2.2 Component Diagram

Figure 2.1 is a simplified diagram that outlines the component interactions:

## 2.3 Message Flow and Data Exchange

### 1. Input Submission:

The Mission Input Node collects the primary mission data (an array of waypoints and time parameters) and sends it to the Deconfliction Service Node.

### 2. Flight Schedule Retrieval:

The Deconfliction Service queries the Flight Schedule Node, which loads schedules from a JSON file (e.g., `flight_schedules.json`).

### 3. Parallel Conflict Checks:

- **Spatial:**

The spatial conflict checker converts waypoints into 2D line segments using only the x and y coordinates (ignoring altitude for the check) and applies geometric algorithms using Shapely.

- **Temporal:**

The temporal conflict checker converts time strings to seconds since midnight and compares mission time windows for overlaps.

#### 4. **Conflict Aggregation:**

Results from the spatial and temporal checkers are aggregated and a final clearance status is generated. If conflicts exist, detailed information (locations, conflict distances, and affected drone IDs) is provided.

#### 5. **Visualization:**

The Visualization Node creates graphical outputs (static images or animations) showing the primary mission path, conflicting trajectories, and the specific conflict points.

## CORE IMPLEMENTATION

### 3.1 Flight Schedule Node

**File:** flight\_schedule\_node.py

**Role:** Acts as a centralized database for simulated flight schedules. The node reads from a JSON file, sanitizes the data, and provides structured flight schedules to the deconfliction service.

```
for drone_id, schedule_data in schedules_data.items():
    waypoints = []
    for wp in schedule_data['waypoints']:
        waypoints.append((wp['x'], wp['y'], wp.get('z', 0.0)))

    start_time = datetime.strptime(schedule_data['start_time'], '%H:%M:%S')
    end_time = datetime.strptime(schedule_data['end_time'], '%H:%M:%S')

    schedules[drone_id] = {
        'waypoints': waypoints,
        'start_time': start_time,
        'end_time': end_time
    }
```

Figure 3.1: Key Code Excerpt

Data Representation: A `FlightSchedule` object contains:

- **drone\_id:** Unique identifier.
- **waypoints:** List of tuples representing coordinates.
- **start\_time/end\_time:** Times converted to seconds.

### 3.2 Deconfliction Service Node

**File:** deconfliction\_service\_node.py

**Function:** Coordinates the overall deconfliction process by concurrently invoking spatial and temporal conflict checks.

```
def get_mission_clearance_callback(self, request, response):
    spatial = self._check_spatial_conflicts(request.waypoints)
    temporal = self._check_temporal_conflicts(request.start_time, request.end_time)
    response.is_clear = not (spatial['has_conflicts'] or temporal['has_conflicts'])
    return response
```

Figure 3.2: Orchestration Example

This node bridges the mission input and the conflict checkers, ensuring that the final response is a consolidated view of all potential conflicts.

### 3.3 Spatial Conflict Checker Node

**File:** spatial\_conflict\_checker\_node.py

#### Algorithm

- **Waypoint Processing:**

Converts input waypoints (which may be received as dictionaries or flat arrays) into a standardized flat list. If only 2D data is provided, it appends a default  $z$ -value of 0.0.

- **Line Segment Construction:**

Uses Shapely to build 2D line segments from consecutive waypoints.

- **Conflict Detection:**

Checks for intersections using `primary_line.intersects(drone_line)` and computes distances. If the distance is less than the safety buffer (5m), it flags a conflict.

```
if primary_line.distance(drone_line) < self.safety_buffer:
    conflict_point = primary_line.intersection(drone_line)
    response.has_conflicts = True
    response.conflict_locations_x.append(float(conflict_point.x))
    response.conflict_locations_y.append(float(conflict_point.y))
```

Figure 3.3: Key Code Expert

#### Edge Cases:

- Zero-length segments are skipped.
- The node handles parallel lines and floating-point precision issues.

### 3.4 Temporal Conflict Checker Node

**File:** `temporal_conflict_checker_node.py`

#### Logic:

- Converts all time strings into a numeric format (seconds since midnight).
- Determines if the mission time window overlaps with any drone's schedule:
- Supports special cases such as missions that span midnight.

```
def _has_temporal_overlap(self, window1, window2):  
    return max(window1[0], window2[0]) < min(window1[1], window2[1])
```

Figure 3.4: Overlap Check Example

### 3.5 Visualization Node

**File:** `visualization_node.py`

#### Features:

- Generates 2D plots using Matplotlib.
- Highlights conflict zones by overlaying markers on the mission path.
- Produces animations that dynamically show the evolution of drone trajectories and conflict points.

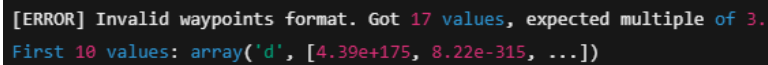
This provides visual feedback and helps in debugging and demonstration of the system.

```
def animate(frame):  
    ax.clear()  
    ax.plot(primary_path_x, primary_path_y, 'b-')  
    ax.plot(drone_path_x, drone_path_y, 'r--')  
    if conflict_at(frame):  
        ax.add_patch(Circle(conflict_loc, radius=5, color='red', alpha=0.3))
```

Figure 3.5: Visualization Code Snippet

## ANALYSIS OF LIMITATIONS AND ERROR INVESTIGATION

### 4.1 Memory Corruption and Waypoints Format Error



```
[ERROR] Invalid waypoints format. Got 17 values, expected multiple of 3.  
First 10 values: array('d', [4.39e+175, 8.22e-315, ...])
```

Figure 4.1: Error observed

#### Root cause

##### 1. Uninitialized Memory Access:

The extreme values (e.g.,  $4.39e + 175$ ) are indicative of reading unallocated or corrupted memory. This is symptomatic of low-level issues in how the ROS 2 message serialization is handled between nodes.

##### 2. Serialization Bug:

A mismatch in the ROS 2 message definitions can lead to type confusion. For example, using `float[]` instead of `float64[]` may cause the system to treat the data incorrectly, leading to array size mismatches.

##### 3. Validation Deficiency:

There was insufficient validation when extracting waypoint data from the incoming message. The error surfaces when the system expects a flat list of coordinate values (multiples of three) but instead encounters a mix or an incomplete set of values.

#### Debugging Steps:

- **Data Validation:**

A proposed fix was to introduce a validation function

```
def validate_waypoints(waypoints):  
    return len(waypoints) % 3 == 0 and all(-1e6 < x < 1e6 for x in waypoints)
```

Figure 4.2: validation function

- **Message Definition Review:**

Reviewing the ROS 2 service definition to ensure that the waypoint array is declared as float64[] rather than a less precise type.

- **Bounds Checking:**

Adding additional bounds checking before processing the waypoints to avoid interpreting uninitialized memory.

## 4.2 Other Constraints and Unimplemented Features

### Time Constraints:

Due to the limited development window (48 hours), certain enhancements had to be postponed:

- **Complete 4D Conflict Resolution:**

While 3D spatial checks were implemented, full altitude-time interaction handling remains partially complete.

- **Dynamic Replanning:**

A real-time mission replanning module was considered but not implemented.

- **Performance Optimization:**

The current spatial checks are  $O(n^2)$  and might not scale well with many UAVs. Optimizations such as R-tree or quadtree indexing are future improvements.

### Lessons Learned:

- **ROS 2 Service Timeouts:**

Robust handling of service timeouts and data serialization issues is critical.



- **Precision in Message Definitions:**

Explicit use of float64[] in ROS 2 interfaces can mitigate subtle memory corruption bugs.

- **Python Limitations:**

The Global Interpreter Lock (GIL) and performance characteristics of Python pose challenges in high-concurrency scenarios.

## CONCLUSION AND FUTURE WORK

### 5.1 Key Achievements

- **Modular Architecture:**

Developed a ROS 2 service-based system with dedicated nodes for spatial and temporal conflict checking.

- **Robust Conflict Detection:**

Implemented mechanisms to detect both spatial intersections and temporal overlaps between UAV trajectories.

- **Visualization Pipeline:**

Created animated and static visual outputs to demonstrate the system's functionality.

### 5.2 Lesson learned

- The importance of rigorous data validation and explicit type declarations in inter-node communications.
- Challenges associated with rapid prototyping under strict time constraints, and how some design choices may lead to memory access issues.
- The need for advanced indexing and optimization strategies when scaling up to real-world applications with thousands of UAVs.

### 5.3 Future Enhancements

- **Complete 4D Resolution:**

Fully integrate altitude ( $z$ ) and time into the conflict detection logic.

- **Optimized Spatial Queries:**

Incorporate R-tree or quadtree structures to reduce the complexity of spatial checks.

- **Dynamic Replanning:**

Develop a module for real-time mission adjustment based on detected conflicts.

- **Extended Validation and Debugging:**

Enhance ROS 2 message definitions and incorporate memory sanitization tools (e.g., AddressSanitizer) to prevent uninitialized memory access.

- **Advanced Visualization:**

Expand the visualization module to support interactive 4D plotting using libraries like Plotly.