UAV Strategic Deconfliction System

Technical Documentation

1. Overview

This document provides a detailed technical overview of the UAV Strategic Deconfliction System. The system is a full-stack application designed to verify whether a drone's planned waypoint mission is safe to execute in a shared airspace populated by other simulated aircraft. It provides a real-time, 4D (3D space + time) analysis and an interactive user interface for mission planning and visualization.

2. System Architecture

The application is built on a modern client-server architecture, separating the core deconfliction logic on the backend from the interactive visualization on the frontend.

- **Backend:** A Python-based server built with **Flask** and **Flask-SocketIO**. It exposes a REST API for on-demand conflict checks and uses WebSockets to stream live simulation data.
- **Frontend:** A single-page web application (mission_control.html) built with standard HTML, JavaScript, and styled with **Tailwind CSS**. It uses the **Three.js** library to render an interactive 3D airspace.
- **Data Flow:**
    1. The frontend sends a proposed mission plan to the backend via an HTTP POST request.
    2. The backend's deconfliction engine processes the request and returns a JSON response indicating "CLEAR" or "CONFLICT".
    3. Separately, the backend can run a continuous simulation, streaming the real-time positions of all drones to the frontend via WebSockets.

3. File Breakdown

The project is organized into a logical directory structure to separate concerns.

data/

This directory contains all static data and configuration.

- config.json: Stores system-wide parameters, such as the default safety buffer and the performance profiles (speed) of different drone models.
- scenarios.json: Contains one or more simulation scenarios, each defining a primary mission and a list of simulated background flights with their fixed trajectories.

src/

This directory contains the core backend logic.

- data_models.py: Defines the core data structures (Waypoint, PrimaryMission, SimulatedFlight) using Python's dataclasses. Each class includes __post_init__ validation to reject invalid data at the point of creation.
- deconfliction_logic.py: The heart of the system. It contains the **Hybrid Deconfliction Engine**, which uses a combination of a broad-phase filter and a narrow-phase check.
- utils.py: Contains helper functions, most notably a vectorized NumPy function to calculate mission ETAs efficiently.
- visualization.py: A module to generate animated .gif files of scenarios from the command line, fulfilling a secondary project requirement.

tests/

This directory holds the automated testing suite.

- test_data_models.py: Contains unit tests that verify the validation logic in the data models.
- test_advanced_logic.py: Implements a property-based fuzz test using the Hypothesis library to rigorously validate the correctness of the deconfliction algorithm against thousands of random scenarios.
- benchmark_*.py: A suite of benchmark tests to provide quantitative proof of the hybrid engine's performance at scale.
- fuzzy_test.py: A targeted fuzz test script to ensure robustness against malformed and corrupted data inputs.
- test_integration.py & test_server.py: Integration tests to verify the interaction between different system components and the API endpoints.

Root Directory

- server.py: The entry point for the interactive web application. It starts the Flask server, defines API endpoints, and handles WebSocket communication.
- main.py: The entry point for the command-line version of the tool. It uses argparse to run scenarios, making it suitable for automated testing and scripting.
- mission_control.html: The self-contained frontend application that provides the 3D visualization and user interface.
- requirements.txt: Lists all Python dependencies with pinned versions for a reproducible environment.

4. Testing Strategy & Validation

A multi-layered testing strategy was implemented to ensure the system's correctness,

robustness, and performance.

Test Suite Summary

The complete test suite is executed using `pytest`. The suite consists of **25 tests** that cover all aspects of the backend system, from individual functions to full integration. All 25 tests pass successfully, confirming the stability and correctness of the application.

Unit Testing

- **Objective:** To verify the integrity of the basic data structures.
- **Implementation:** Using pytest, we wrote specific tests for the classes in data_models.py. These tests confirm that creating an object with invalid data (e.g., a PrimaryMission with start_time >= end_time) correctly raises a ValueError.

Property-Based Testing

- **Objective:** To rigorously validate the mathematical correctness of the core deconfliction algorithm against a vast number of inputs.
- **Implementation:** We used the **Hypothesis** library (test_advanced_logic.py) to create a property-based test. Instead of fixed examples, this test generates hundreds of random scenarios, including countless edge cases that would be impossible to design manually. It asserts two critical properties:
    1. **No False Positives:** If the engine reports a conflict, the distance between the drones at the reported time *must* be less than the safety buffer.
    2. **No False Negatives:** If the engine reports "CLEAR," a trusted internal calculation confirms that the minimum distance was *never* less than the safety buffer.

Robustness & Fuzz Testing

- **Objective:** To ensure the system handles malformed and corrupted data gracefully without crashing.
- **Implementation:** A dedicated fuzz test script (fuzzy_test.py) was created to submit invalid data structures to the deconfliction engine, including missions with mismatched list lengths, backward timestamps, and corrupted data types.
- **Result:** All fuzz tests passed, confirming the system is robust against unexpected inputs.

Benchmark & Scalability Testing

- **Objective:** To provide quantitative proof of the system's performance at scale and validate our architectural choices.

- **Implementation:** We conducted several benchmark tests, including a "swarm test" with a high number of simulated flights. The profiler output shows that the vast majority of the execution time is spent in the add_flight function of our Grid4D class, which is our highly optimized broad-phase filter.
- **Results:** The system demonstrates excellent performance, handling large-scale scenarios efficiently.
    - **10,001 Flights:** The deconfliction check completed in **0.111 seconds**.
    - **50,001 Flights:** The deconfliction check completed in **0.892 seconds**.

This proves the scalability of the hybrid engine design, as it can process tens of thousands of potential conflicts in under a second.

Edge Cases Handled

Our design and testing strategy explicitly handles numerous edge cases, including:

- Stationary drones (primary or simulated).
- Perfectly parallel or collinear flight paths.
- Missions with only two waypoints.
- Missions that are physically impossible to complete within the specified time window.
- Invalid data inputs from configuration files or API requests.
- Zero-duration flight segments where start and end times are identical.