Reflection & Justification: UAV Strategic Deconfliction System

1. Design Decisions and Architectural Choices

For this project my main goals were to keep the code modular and flexible enough to scale if needed. To do that, I split the work across different modules:

1. **data_model.py:** Holds the basic building blocks like Waypoint, Flight, and Conflict.
2. **trajectory.py:** Takes care of interpolation between waypoints. Right now, it's linear interpolation, but I left room for smoother methods like splines if needed later.
3. **collision_check.py:** This is where the conflict detection happens. I added two ways to check:
    a. Sampled check: Step through time and see where drones are at each moment. **(Basic Idea)**
    b. Analytic check: Directly calculate the closest approach between flight paths. **(AI generated)**
4. **simulator.py:** Generates scenarios. I made it create both random and hardcoded flights so I could test all outcomes.
5. **visualize.py:** Handles plotting in 2D and 3D.
6. **cli_api.py:** A simple entry point where you can run a mission check without going into the lower-level code.
7. **main.py**: It calls the simulator to generate flights, runs the conflict detection, and then shows the results. I used it to test both random scenarios and hard-coded ones. (Both 2D and 3D).

2. Spatial and Temporal Checks

Conflicts are determined by looking at both where the drones are and when they are there.

1. **Spatial Check:**
    a. The code measures the distance between drones either at discrete time steps or using analytical method.
    b. In the analytic method, it computes the exact closest distance between two straight-line segments by considering each segment's start and end positions over time, and by identifying the location and time when the drones come nearest to each other. If that distance is less than the buffer, it is considered as a collision.

2. **Temporal Check:** Each waypoint has a time offset from the start of the mission, which lets the system figure out where a drone is at any given moment. Conflicts are only considered if the drones time windows overlap.

The buffer distance sets the minimum safe distance between drones. If two flights get closer than this distance at the same time, a Conflict object is created that records which flights were involved, when and where it happened, and how close the drones came.

3. AI Integration

My work:

1. I implemented all the core functionality, defining flights and waypoints, computing trajectories, detecting conflicts, running simulations, and visualizing results in 2D as well as 3D.
2. The core logic for checking safety, handling time and space overlaps, and managing conflicts was written by me.

AI contribution:

1. AI was used to refine the code, improve docstrings, and suggest some advanced approaches like the analytic closest-approach method.
2. AI suggestions were applied where helpful, but the overall structure and core logic were created and implemented by me.

AI Evaluation:

1. I reviewed all suggestions made by AI and tested them in different flight scenarios to make sure they actually worked.
2. If something didn’t fit or wasn’t quite right, I tweaked it or rewrote it myself to make sure it matched the way the code should work.

4. Testing Strategy

I used pytest to check that the system works correctly in different situations (tests/test_deconfliction.py):

1. Conflict-Free Flights: Flights that are far apart in space or time to make sure the code reports SAFE.
2. Single Conflict: Two flights that cross paths to verify the code detects the conflict and reports the correct distance and time.
3. Multiple Conflicts: Three or more flights with overlapping paths to test handling of multiple conflicts at once.

Edge Cases

1. Non-overlapping time windows: Flights that share the same space but happen at different times don't count as conflicts.

```
        # Get position of primary flight
        primary_pos = get_position_at(primary, current_time)
        if not primary_pos:
            t += time_step
            continue
```

2. Missing time offsets: Waypoints without explicit time offsets are automatically handled in the get_position_at function.

```
    waypoints = flight.waypoints
    times = [wp.time_offset if wp.time_offset is not None else idx for idx, wp in
enumerate(waypoints)]
```

3. Simultaneous start or end times: Flights starting or ending at the same moment are checked accurately.

```
    # Compute total duration of flight
    total_duration = primary.mission_window.duration()
    t = 0.0

    # Go through entire time duration to check conflicts
    while t <= total_duration:
```

4. Altitude only conflicts: Differences in height are considered using full 3D distance calculations.

```
# Function to calculate distance between 2 points
def euclidean_distance(p1, p2) -> float:
    """Compute 3D Euclidean distance between two points."""
    return np.sqrt((p1[0] - p2[0])**2 +
                   (p1[1] - p2[1])**2 +
                   (p1[2] - p2[2])**2)
```

5. Scalability Considerations

To scale this system for real-world operations with thousands of drones:

1. **Data handling:** Organize flight positions using spatial structures like k-d trees or R-trees, so we don’t have to check every pair of drones. **(AI generated)**
2. **Time filtering:** Only check flights that are in the same time window, instead of comparing all flights at all times.
3. **Parallel computation:** Use multiple CPU cores or GPUs to check conflicts for many flights at once.
4. **Dynamic deconfliction:** Accept real-time drone positions, not just waypoints, so conflicts can be detected as drones move.
5. Efficient storage: Keep current and predicted flight positions in a fast in-memory database to quickly check distances between drones. **(AI generated)**