1. Analyze requirements to determine appropriate testing strategies

1.1 Range of Requirements: Functional Requirements, Measurable Quality Attributes, Qualitative Requirements

The primary functional requirement of the ILP delivery system is to compute a valid and efficient drone delivery path from a restaurant to Appleton Tower, ensuring compliance with central area constraints and avoiding no-fly zones. This requirement necessitates robust validation of input data, precise navigation logic, and reliable interaction with external data sources such as restaurant locations and no-fly zones. For instance, the OrderValidation class ensures that input orders meet the expected constraints before processing. The validate() method checks critical aspects such as:

- **Pizza definitions and pricing validity** (isPizzaDefined(), isPizzaPriceValid())
- **Credit card validity** (isValidCreditCard())
- **Order date validity** to prevent outdated transactions
- **Ensuring a single restaurant source for the order** (findMatchingRestaurant())

The measurable quality attributes include performance efficiency (minimizing request response time), correctness (generating a valid, optimal path), and robustness (handling invalid orders gracefully). Performance testing was crucial to detect and optimize slow calculations, as seen in early tests where the initial calculatePath() method took over 400ms on the first execution due to API fetching overhead.
To improve response time, the optimizations included:

- **Reducing console logging** in calculatePath()
- **Minimizing redundant API calls** by caching fetched data
- **Refactoring direction adjustments** to prioritize efficient pathing

1.2 Level of Requirements: System, Integration, Unit

The ILP delivery system was tested at **multiple levels** to ensure correctness:

- **Unit Tests:** Focused on individual methods in OrderValidation, LngLat, and DeliveryController. Examples include:
  - Validating closeTo() in LngLat to check for convergence criteria.
  - Ensuring isInsideCentralArea() correctly identifies valid central area coordinates.
  - Testing intersectsNoFlyZone() to verify that paths correctly avoid restricted areas.
- **Integration Tests:** Ensured smooth interaction between components such as:
  - **OrderValidation** and external restaurant APIs.
  - **calculatePath()** and adjustDirection() logic to ensure the drone follows a **legal** and **optimal** route.
- **System-Level Testing:** The final **Postman tests** simulated **real-world API calls**, sending JSON requests representing different orders and verifying the computed paths.

1.3 Identifying Test Approach for Chosen Attributes

To ensure comprehensive coverage, the following testing strategies were employed:

- **Equivalence Partitioning & Boundary Value Analysis**
  - Tested **valid** and **invalid** orders to ensure correct 200/400 responses.
  - Verified the handling of credit card expiry dates (past, present, future cases).
  - Checked order dates to ensure system rejects outdated orders.
2. **Path Coverage Testing**
   - Ensured multiple restaurants had valid, direct paths (Halal Pizza, Domino's).
   - Tested cases where no-fly zones forced detours (Civerinos Slice).
   - Logged step-by-step movements in calculatePath() to manually verify correctness.
3. **Performance Testing**
   - Measured first-request latency (~400ms) and optimized API calls.
   - Reduced execution time by removing unnecessary logging and optimizing direction calculations.
4. **Regression Testing**
   - After every optimization (e.g., caching no-fly zones), previously validated tests were re-run to ensure no unintended failures occurred.

1.4 Assess the Appropriateness of Your Chosen Testing Approach

The selected test strategies effectively validated both functional correctness and performance efficiency. The testing approach was appropriate because:

- **Unit testing** ensured critical methods (e.g., closeTo(), isInsideCentralArea()) functioned as expected.
- **Integration testing** verified smooth data flow between OrderValidation, DeliveryController, and API dependencies.
- **Path testing with real-world scenarios** confirmed no-fly zones were avoided correctly while maintaining efficient routing.
- **Performance monitoring** led to reduced request times by optimizing log output and refining path calculations.

However, some limitations were noted:

- The initial response time was still higher than ideal (~400ms on first request). A persistent cache for central area/no-fly zones could further improve speed.
- Real-world drone navigation tests were outside the project scope, meaning physical-world variables (e.g., wind resistance) were not considered.

Moving forward, additional automated performance benchmarking could help ensure continued optimization and scalability.