Learning Outcome 2: Design and Implement Comprehensive Test Plans with Instrumented Code

2.1 Construction of the Test Plan

The development of a test plan began with understanding the requirements and constraints of the system. Given the predefined list of orders, it was apparent that unit testing could be effectively employed to validate order processing logic. Each test case was designed to evaluate specific conditions, such as valid orders, orders with errors, and edge cases.

The need for seamless integration between the OrderValidationService and the CalcDeliveryPath service was a key consideration in the test plan. To ensure this, a robust OrderValidation mechanism was implemented early in the development process. This allowed the validation results to serve as a dependable input for path calculation, ensuring that only valid orders proceeded to pathfinding.

Not all requirements were tested programmatically, such validating paths for CalcDeliveryPathAsGeoJson in geojson.io. some tests were verified manually because it would create unnecessary complexion to do unit testing for these with no benefit to verifying compliance with requirements.

Test Plan

Functional	**How its being tested**	**When tests fail**
Calculate distance, Are distances close	Unit testing with coordinates to test core functionality	-when invalid input is inserted(null values or invalid coordinates) -the wrong error code is returned
Next position calculation	unit testing with coordinates and angle	-invalid input, null values, edge cases
Is in region	unit testing with a closed region and position	-Non-closed regions, invalid input
Validate order	unit testing using array of orders	valid orders return invalid and vice versa. System returns wrong error code.
ensure paths don't leave the central area and don't go in no-fly zones	Unit testing checking if any paths don't adhere to the rules specified.	-when paths leave central area -when paths go through a no fly zone - when a drone does too many moves
Return geojson path	Pasting the output of calcDeliveryPathAsGeoJson into geojson.io	-output is not in the correct format and therefore not able to paste it into a mapping tool
Non-functional		
Meaningful error messages	running endpoints	-gives a bad status code or incorrect error message
Low runtime	running endpoint for calcdeliverypath	-when tests take over 60s
Runnable from docker image	packaging and running	-system does not run correctly
Path accuracy	manually	if paths take non-optimal routes

2.2 Evaluation of the Quality of the Test Plan

The test plan was evaluated based on its comprehensiveness and alignment with the defined requirements. Key metrics used for evaluation included:

- **Coverage of Requirements:** All requirements, including order validation, pathfinding, and integration, were addressed in the test plan. Edge cases were also thoroughly tested successfully to ensure the system's robustness. The most tested parts of the code were the Ordervalidation and CalcDeliveryPath endpoints since they were the most complex and were worth the most marks for this project.
- **Measurability of Testing:** The tests were designed to produce clear and measurable outcomes, making it easy to identify and address issues. For example, runtime metrics for pathfinding were collected to assess and optimise performance. Error messages for other endpoints were also used to ensure they worked for valid and invalid inputs.
- **Effectiveness:** The test plan effectively identified issues in the early stages of development, reducing debugging efforts later in the process. This approach ensured that the final system met both functional and non-functional requirements.

The quality of the test plan was further demonstrated by its ability to guide the development process and validate the system comprehensively against the coursework's requirements. Although the testing plan was not perfect, with it being heavily skewed towards unit testing and a limited amount system and integration testing was done. This being said the docker image was heavily tested as it was essential to have a working docker image for any of the system to even be run by an instructor of the ILP course.

2.3 Instrumentation of the Code

Postman:

- **Purpose**: Postman was utilized to test API endpoints, focusing on runtime performance and response status codes.
- **Integration**:
    - Each endpoint in the application, such as /validateOrder, /distanceTo, and /calcDeliveryPath, was tested using Postman.
        - **Test cases** included:

          Confirming HTTP status codes for various scenarios (e.g., 200 OK for successful operations, 400 BAD REQUEST for invalid inputs).

          Measuring runtime for API responses.

Temporary Debugging

- **Purpose**: Temporary logging statements were added to identify issues during development and testing.
- **Integration**:
    - Logging statements were placed in critical parts of the application:
        - Key service methods like validateOrder and findPath in CalcDeliveryPath to log input data, intermediate steps, and results.

        - Once issues were resolved, these statements were removed.

2.4 Evaluation of the Instrumentation

Postman:

- Effectiveness:
    - Postman proved to be an invaluable tool for systematically testing API endpoints. It provided clear insights into runtime performance and helped validate status codes against expected outputs.
    - The structured test cases ensured comprehensive coverage of both typical and edge-case scenarios. For example:
        - Sending invalid data to endpoints like /validateOrder effectively confirmed proper handling of 400 BAD REQUEST errors.
- Limitations:
    - While Postman identified runtime and response-related issues, it lacked deeper visibility into code execution paths or memory usage.

Temporary Debugging:

- Effectiveness:
    - Temporary logging provided quick and direct insights into the application's execution flow, making it easier to pinpoint issues during development.
    - Logs placed in critical areas, such as validateOrder and findPath, revealed issues like invalid input handling and inefficient pathfinding logic.
    - This approach allowed rapid feedback and resolution during early development stages without requiring external debugging tools.
- Limitations:
    - Temporary logs cluttered the codebase, and ensuring all were removed before production added additional overhead.
    - Logs did not provide structured data for post-analysis, making it harder to derive insights over time compared to using logging frameworks with centralized monitoring.