Learning Outcome 4: Limitations of Testing Process

4.1 Testing Deficiencies

Lack of Stress Testing & Concurrency Testing:

Currently our testing primarily focuses on single-order validation and processing. The 600 test cases run sequentially, not simulating real-world concurrent loads. During peak hours, such as Lunch, we can expect many users to be using the service at once. This could have disastrous consequences, including:

- Possible drone path conflicts when multiple deliveries are scheduled simultaneously
- Potential race conditions in order processing remain undetected
- Path calculation performance degradation with multiple concurrent requests

Code Quantity Overshadows Code Quality:

As it currently stands, we have over 95% code coverage. Additionally, over 600 hand-crafted test cases were programmatically created and utilised. These impressive numbers give a false sense of confidence in the system reliability. As a direct result, the following issues likely arose:

- The manual test creation likely missed at least a single edge case that the brainstorming missed.

The solution would be to create more thorough requirements. Currently, a single test can satisfy a requirement spanning some functionality. The ideal is that multiple tests satisfy multiple requirements spanning that functionality.

System Testing Depth

The unit testing in my project is extensive. This is further proven by the extensive code coverage and testing. But the other parts of the testing pyramid are distinctly lacking (apart from the single performance test that mocks a single realistic order). This means many realistic workflows are not getting tested end-to-end. Because of this lack, errors are likely to arise where one service connects to another. After all, they've not been tested as thoroughly.

Testing Procedure	Target	What Happened
Unit Tests	Multiple Tests per functionality	- Multiple tests were successfully created for each functionality. - Almost all of the small atomic units of the application were thoroughly tested, with each receiving an individual test.
Integration Tests	Integration test *between* every service	- Mock requests were created and sent through to the application. - There was a specific emphasis on tests with malformed input. - There was insufficient testing when it came to how internal services communicated with each other (not including the HTTP requests with the main controller).
System Tests	3+ Tests simulating specific scenarios end-to-end	- There was a single system test evaluating the system end-to-end (performance test).
Coverage	80%+	- Code Coverage was 95%. This is highly commendable and well above industry standard practise (of 80%).

4.4 What's Required to Achieve Targets?

Testing Procedure	Target	How To Achieve Targets?
Unit Tests	Multiple Tests per functionality	- This was achieved thoroughly with the current implementation. While this was the goal, it also may have provided a false sense of security.
Integration Tests	Integration test *between* every service	- This was partly achieved with the current implementation, with testing between the Controller and every relevant service. - Integration tests between: *DroneService* and *MapPath,* as well as *OrderService* and *RestaurantService*.
System Tests	3+ Tests simulating specific scenarios end-to-end (Programmatically)	- To achieve this target, the following would be implemented: - Multiple malformed HTTP requests. Correct output should be corresponding error. - Multiple valid HTTP Requests. Their output is checked programmatically for validity.
Coverage	80%+	- This was already comfortably achieved. Little more is required for this.