LO3 - Apply a Wide Variety of Testing Techniques and Compute Test Coverage and Yield According to a Variety of Criteria:

3.1 – Range of Techniques:

The testing strategy used a systematic, specification-based approach. This aligns with the functional testing and partition testing principles, where the input space is divided into meaningful equivalence classes to increase the likelihood of detecting faults in failure-prone regions. A testing log can be found in **'docs/test/log.md'**, detailing all tests carried out and defects detected as well as actions/resolutions taken. A full summary detailling all the testing techniques utilised and where and how can be found in **'docs/test/test_techniques.md'**.

Functional (Specification-Based) Testing

Test cases were derived directly from system requirements and specifications, following the partition principle. Inputs were divided into equivalence classes representing valid, boundary, and erroneous conditions.

- **Equivalence Partitioning:**
  Input domains such as order formats, delivery coordinates, and no-fly zone constraints were divided into valid and invalid classes. For example, delivery paths were tested as:
  - Fully valid paths outside restricted zones
  - Paths intersecting no-fly zones
  - Boundary-grazing paths near restricted areas
- **Boundary Value Analysis:**
  - Critical thresholds, such as coordinate precision limits and maximum delivery distances, were tested using values at, just below, and just above defined boundaries. This targeted areas where failures are more likely to occur.

This systematic selection ensured that tests focused on areas within the input space where faults were more likely to occur rather than uniformly sampling the entire input space.

Structural (White-Box) Testing

Structural testing was carried out alongside functional testing, ensuring internal program logic was tested

- **Statement and Branch Coverage:**
  Conditional logic in services such as RestrictedAreaService and DistanceService were tested for both success and failure paths, including error-handling branches.
- **Method Coverage:**
  All public service methods were tested using unit and integration tests to make sure the interfaces work correctly.

This ensured that both missing logic faults and incorrect implementation faults were addressed.

Integration Testing

Integration testing validated data flow and cooperation between system components, such as:

- Order validation and path generation services
- No-fly zone enforcement and route calculation modules

This confirmed that module interactions matched interface specifications and system-level requirements.

Combinatorial and Robustness Testing

A category-based approach tested key inputs like order format, coordinate limits, and system load. Selected combinations focused on common and high-risk cases rather than testing every possible input.

Performance and Stress Testing

Generated workloads were used to simulate and evaluate system behaviour under increasing request volumes. This followed a scaling approach rather than random traffic generation, enabling consistent performance trend analysis and repeatability.

Regression Testing

Regression testing was conducted after code changes to ensure previously validated features, such as GeoJSON generation and no-fly zone validation, continued to function correctly.

3.2 – Evaluation Criteria for the Adequacy of the Testing:

The adequacy of the testing strategy is evaluated using a set of clear criteria that focus on how well the system's code, behaviour, and performance match the stated requirements. These criteria are designed to provide confidence that important faults are likely to be detected within the limits of the project.

- **Code Coverage (Structural Adequacy):**
  - Code coverage is used to check how much of the system's internal logic is exercised by the tests. Statement and branch coverage are measured for critical components such as navigation, order validation, and no-fly zone enforcement.
  - High coverage increases confidence that most decision paths and error-handling logic have been tested. However, coverage alone does not guarantee correctness, as code can be executed without being properly validated against meaningful inputs.
- **Requirement Coverage (Functional Adequacy)**
  - Each test case is linked to one or more system requirements. This ensures that all selected safety, correctness, and performance requirements are explicitly tested rather than assumed to work.
  - Inputs are grouped into valid, boundary, and invalid categories to check that the system behaves correctly across different types of situations. This approach increases the chance of finding faults that commonly occur at system limits.
- **Integration and Interface Coverage**
  - Integration testing is evaluated by checking that data flows correctly between system components and that modules handle both correct and incorrect inputs from each other. This helps identify problems such as mismatched data formats, missing values, or incorrect assumptions between modules that unit tests alone might not detect.

- **Robustness and Error Handling**-
  - The system is tested with invalid requests, missing data, and out of range values. Adequacy is judged by whether the system fails safely, returns clear error messages, and continues operating without causing wider system failures. This ensures the system remains stable even when users or other systems provide unexpected input.
- **Performance Validation**
  - Response times are measured under generated workloads and compared against defined performance limits. This checks whether the system can meet its timing requirements under expected levels of demand. While this does not fully represent real-world conditions, it provides a consistent and repeatable way to assess system performance.
- **Quality of Test Feedback and Logs**
  - Test reports and logs are reviewed to ensure they clearly show which tests passed or failed and why. Adequate testing makes it easy to trace failures back to specific requirements or components.
  - This supports faster debugging and more reliable regression testing during later development stages.

3.3 – Results of Testing:

The results of testing are presented through a structured testing log (**'docs/test/log.md'**) that records test execution outcomes, identified defects, and corrective actions across unit, integration, performance, and system-level testing. Coverage levels can be found at (**'docs/test/coverage.pdf'**)

Unit Testing Results

- Achieved 90–100% coverage across critical classes which concerned tested requirements such tests such as RestrictedAreaService and DistanceService
- Boundary and invalid-input tests revealed early validation gaps, which were corrected through improved input handling

Integration Testing Results

- Confirmed reliable interactions between core modules
- Detected format mismatches in request handling, which were resolved through interface alignment

Performance and Stress Testing Results

- The system maintained acceptable response times under moderate and high request volumes
- Performance trends demonstrated linear scalability within the tested workload range

Robustness Testing Results

- Bad inputs and invalid coordinate values were consistently handled with appropriate error responses (e.g: HTTP 400)
- No critical crashes or undefined behaviour observed

3.4 – Evaluation of Results:

Strengths

- High structural coverage confirms internal logic paths were thoroughly exercised
- Systematic partition-based testing effectively identified boundary and missing-logic faults
- Integration and stress testing validated system behaviour beyond isolated units

Limitations

- Performance testing relied on synthetic workloads, which may not fully represent real-world network latency or hardware variability
- Long-term endurance and memory usage testing were not conducted
- Some exploratory test scenarios remain manual rather than automated

Overall Assessment

The combination of functional partition testing, structural coverage analysis, integration validation, and defect yield tracking provides strong evidence that the system meets its functional correctness, robustness, and performance requirements within a simulated environment.