LO1 - Analyze requirements to determine appropriate testing strategies:

1.1 – Range of Requirements:

The test basis for this system is derived from the system-level requirements documented in **'docs/requirements.pdf'**. These requirements define the required behaviour that testing activities must verify. A diverse range of requirement types is intentionally included to ensure broad test coverage and to justify the use of multiple testing strategies.

Functional Requirements:

The system requirements include functional requirements which define the core behaviour of the system and form the primary test basis for correctness testing. These requirements span across safety, correctness and liveness. Examples include:

- **SR1 - No Fly Compliance:** The system must never generate a delivery path that enters a no-fly zone. This covers safety.
- **SR2 - Corner Correctness:** Paths must use valid turning points and avoid illegal shortcuts. This also covers safety.
- **CR1 - Positioning Accuracy:** All generated positions must lie within an acceptable tolerance of expected coordinates. This covers correctness
- **CR2 - Compass Direction Compliance:** Movement directions must be limited to the defined compass headings. This also covers correctness
- **LR1: Service Health and Readiness:** The system shall provide health check endpoints indicating service operational status. This covers liveness.

These requirements support system, integration, and unit testing by defining observable and testable behaviours.

Measurable Quality Attributes:

Measurable quality requirements are also defined, specifying quantitative constraints that can be validated through objective testing. The defined measurable requirements covers performance, reliability and maintainability.

Examples include:

- **PR1 - Pathfinding Response Time:** Path generation must complete within 2 seconds for typical Edinburgh routes. This is a performance requirement.
- **RR1 - Input Validation Completeness:** All malformed or invalid inputs must be detected and rejected. This is a reliability requirement.

These requirements justify performance testing, timing measurements, and structured negative testing.

Qualitative Requirements

The requirements defined also cover qualitative requirements, describing desired characteristics that are not always directly measurable but still guide testing decisions. These requirements cover areas such as maintainability, usability, testability and robustness.

Examples include:

- **UR1 – API Error Message Clarity:** Error responses should be understandable and consistent.
- Maintainability and testability requirements that influence modular design and isolation of components.
- Robustness requirements that expect acceptable behaviour under abnormal or degraded conditions, i.e: **ROR2: External Dependency Isolation -** Failure of the ILP REST API shall not crash the system.

Although qualitative, these requirements still contribute to the test basis by shaping expected behaviour and guiding test design.

The above shows that a diverse range of requirements have been identified, considered and defined.

1.2 – Level of Requirements:

The requirements in docs/requirements.pdf are intentionally analysed across multiple levels of concern, allowing testing to be planned at system, integration, and unit levels.

System-Level Requirements

The requirements document covers system level requirements which describe and define the end-to-end behaviour of the complete system.

Examples include:

- **CR6 - Path Validity:** Requires the system as a whole to generate a valid delivery route.
- **SR1 - No Fly Compliance:** Validated only when all subsystems operate together.

These requirements form the basis for system testing that validates complete workflows and externally visible behaviour.

Integration-Level Requirements

The requirements also cover the integration level and these focus on interactions between subsystems that must cooperate correctly.

Examples include:

- The integration between pathfinding logic and no-fly zone validation (SR1).
- The interaction between input validation, path generation, and output formatting.

These requirements justify integration testing that verifies data flow, interface contracts, and correct sequencing between components.

Unit-Level Requirements

Finally, unit-level requirements are also defined which help target isolated components with minimal dependencies.

Examples include:

- **CR4 - Compass Direction Compliance:** Validated by unit tests on direction-calculation logic.
- Validation functions that reject malformed coordinates or illegal moves.

Although implementation is not yet complete, the requirements allow anticipated unit tests to be identified and refined later.

1.3 - Identifying Test Approaches for Attributes:

Based on the defined test basis and required behaviour, a range of testing approaches and techniques is selected to align with specific requirement types and levels.

Unit Testing

Unit testing is used to validate the functional correctness of individual components in isolation, primarily using structural (white-box) and systematic functional testing techniques.

Applicable techniques and requirements include:

- **Boundary value analysis and equivalence partitioning** for Input Validation Completeness (RR1), verifying correct handling of invalid coordinates, missing fields, and out-of-range values.
- **Branch and decision coverage testing** for Compass Direction Compliance (CR4) and Positioning Accuracy (CR1), ensuring all permitted compass directions and distance-calculation paths are exercised.
- **Assertion-based testing** for low-level geometric logic, such as move distance precision and coordinate updates, supporting Move Distance Precision (CR5).

These tests provide fast feedback, support early fault detection, and ensure deterministic behaviour of core algorithmic components.

Integration Testing

Integration testing validates correct interactions between subsystems and focuses on interface behaviour and data flow between components.

Applicable techniques and requirements include:

- **Interface testing** between pathfinding and airspace validation logic for No-Fly Zone Compliance (SR1), ensuring generated paths are consistently checked against restricted areas.
- **Scenario-based integration testing** for Corner Crossing Prevention (SR2), validating that geometric corner cases are handled correctly when multiple subsystems interact.
- **Data-flow testing** across order parsing, validation, and path generation pipelines, supporting Order Validation Completeness (CR2) and Multi-Order Batch Assignment (CR7).

These tests reduce the risk of interface mismatches, hidden integration faults, and inconsistent enforcement of business rules.

System Testing

System testing validates complete end-to-end system behaviour against system-level requirements using **black-box functional testing**.

Applicable techniques and requirements include:

- **End-to-end functional testing** of Path Validity Complete Validation (CR6), verifying that full delivery cycles satisfy all geometric, safety, and capacity constraints.
- **Safety testing** to confirm global enforcement of No-Fly Zone Compliance (SR1) and Return to Base Requirement (SR5) across the entire system.
- **Service availability testing** for Service Health and Readiness (LR1), ensuring the system reports operational readiness under normal startup conditions.

System tests provide confidence that the system satisfies its external obligations and behaves correctly from a user perspective.

Performance and Robustness Testing

Performance and robustness testing address measurable quality attributes and system resilience.

Applicable techniques and requirements include:

- **Microbenchmarking and timed integration tests** for Pathfinding Response Time (PR1), measuring response time under representative synthetic workloads.
- **Negative and stress testing** for robustness-related requirements such as Graceful Failure (ROR1), ensuring constraint violations result in controlled and informative error responses.
- **Fault simulation using mocks** for External Dependency Isolation (ROR2), observing system behaviour when dependent services are unavailable.

These approaches ensure acceptable non-functional behaviour while acknowledging that performance testing provides indicative results rather than statistical guarantees across all operational conditions.

1.4 – Assessing Testing Approaches

Overall, selected testing approaches are well matched to the system's requirements, risk profile, and constraints. Unit and structural testing help provide strong assurance for core safety-critical logic such as geometric calculations, compass direction enforcement, and validation rules. Integration testing is appropriate for detecting interaction faults between subsystems such as pathfinding, no-fly zone enforcement, and order validation, which contain major risks. System testing validates end-to-end behaviour against external requirements, ensuring that complete delivery cycles satisfy safety, correctness, and liveness constraints. Performance and robustness testing address measurable quality attributes and failure handling. However, these approaches are necessarily limited by the use of synthetic workloads and mocked dependencies which may not reflect the variability of a real-world environment. These limitations are understood and accepted, as the testing strategy prioritises high-risk functional and safety requirements while remaining feasible.

Testing Approach	Advantage	Disadvantage
Unit Testing	- Enables early detection of defects in core logic - High coverage of algorithms (e.g., Compass Direction Compliance, Move Distance Precision)	- Cannot detect faults resulting from component interaction - Does not validate end-to-end system behaviour

	- Fast to execute and easy to automate	
Integration Testing	- Detects errors between subsystems - Requirements which concern multiple components require integration testing (e.g., pathfinding and validation) - Reduces risk of late-stage system failures	- More complex to design and maintain - May not fully reflect real-world environment due to mocked dependencies
System Testing	- Vanlidates complete end-to-end behaviour against system-level requirements - Can help evaluate non-functional requirements (i.e: performance, reliability)	- Time-consuming to design and carry out all tests - Exhaustive testing is impractical
Performance Testing	- Provides repeatable and comparable timing measurements	- Produces estimates rather than guaranteed real-world performances
Robustness and Negative Testing	- Ensures graceful failure and clear error handling under invalid inputs - Improves system reliability and user trust	- Can’t guarantee resilience against all unforeseen failure modes - Some rare fault combinations may remain untested