# LO2: Test-Planning Document

Design and implement comprehensive test plans with instrumented code

## Introduction

This test plan is designed to outline an initial testing strategy for the PizzaDronz project based on the functional and non-functional requirements specified in the LO1-Requirements document. As the project evolves, this plan is expected to adapt, following the principles of Test-Driven Design (TDD).

The Agile software development lifecycle model has been selected as the most suitable methodology for this project. Agile's iterative and flexible approach ensures continuous testing and validation throughout development. This dynamic process aligns seamlessly with TDD principles, enabling testing of individual components and features as they are developed, without waiting for the entire system to be completed.

Given resource constraints, it is impractical to address every requirement during the initial testing phase. Instead, this plan prioritizes critical functional requirements and robustness-related non-functional requirements. Performance and efficiency testing will be allocated fewer resources but remain essential for final system evaluation. This document provides a general testing strategy for all requirements and detailed test plans for selected high-priority requirements.

## Priority and Prerequisites

1. Functional Requirements

- High Priority: All system-level functional requirements, as outlined in the LO1-Requirements document, must take precedence as they define the core application functionality. This includes lower-level integration and unit requirements that directly contribute to fulfilling these system-level requirements.
- Early Validation: To minimize testing and debugging overhead later in the project, Verification and Validation activities should begin at the earliest development stages. Unit tests will be fully designed in advance, while integration and system-level tests will evolve iteratively in the Agile lifecycle as new functionality is developed and validated.

2. Non-Functional Requirements

- Performance and Efficiency: These requirements are lower priority compared to functional ones, as they do not directly impact core functionalities like drone movement or order validation. Testing will focus on final system optimization.
- Robustness: This is a high-priority non-functional requirement. Early testing will simulate error scenarios to ensure that the system can handle unexpected situations gracefully.
- Timing: While robustness testing can be addressed early, performance and efficiency will primarily be validated during the integration and system testing phases.

General Procedure

1. Unit Testing: Conducted during development for individual components.
2. Integration Testing: Performed after individual components have been validated via unit tests. This will test interactions between components, such as order validation with restaurant menu data.
3. System Testing: Conducted once the entire application is built to validate system-wide behaviour and performance.

At any stage, if tests fail, we will address the issues and rerun tests until the requirement is satisfied. This iterative process is central to Agile and TDD methodologies.

## Detailed Test Plan for Selected Requirements

Requirement 1: Input Validation

Description: The application must only accept the date selected for output files in the format YYYY-MM-DD and a valid base URL for the data fetching.

Testing Strategy:

- Use a variety of correct and incorrect input combinations to validate the application's ability to accept or reject arguments as per the requirements.
- Test cases:
    - Valid inputs: Two arguments (correct date, URL).
    - Invalid inputs: Missing arguments, incorrect formats, or invalid URLs.

Scaffolding and Instrumentation:

- No scaffolding is required for this requirement.

- Use assertions and logging to confirm that inputs are correctly validated.

Requirement 2: Order Validation

Description: Each order must meet a set of detailed criteria for validity, including customer name, order number, payment details, and total cost.

Testing Strategy:

1. Unit Testing:
    - Validate individual criteria using diverse test cases:
        - Correct customer name (at least 2 characters).
        - Order number format (8-character hexadecimal string).
        - Valid credit card details (number, expiry date, and CVV).
        - Pizza details (names match the menu, and all pizzas come from the same restaurant).
    - Simulate edge cases, such as invalid credit card details or mismatched restaurant menus.
2. Integration Testing:
    - Mock orders comprising all components to validate the system's ability to check all criteria holistically.

Scaffolding and Instrumentation:

- Mock restaurant database with menu details (pizza names and prices).
- Mock order database to simulate diverse scenarios.
- Assertions for each validation criterion to ensure accurate results.

Requirement 3: Outcome Recording

Description: The application must record outcomes for all orders based on their validity and delivery status.

Testing Strategy:

- Test the association of orders with all possible outcomes except "DELIVERED" (due to limited resources for flightpath testing in this phase).

- Mock various order scenarios to test outcomes like INVALID_CARD_NUMBER, INVALID_EXPIRY_DATE, and VALID_BUT_NOT_DELIVERED.

Scaffolding and Instrumentation:

- Use the same mock databases as R2.
- Generate synthetic orders to simulate each possible outcome.

Requirement 4: Robustness

Description: Ensure the system remains operational in the face of unexpected problems, such as missing server data.

Testing Strategy:

- Simulate scenarios where:
    - Central area or no-fly zone data is missing (system should run).
    - Restaurant or order data is missing (system should terminate gracefully).
- Test exception handling for server communication errors and invalid inputs.

Scaffolding and Instrumentation:

- Simulated REST server to replicate data availability scenarios.
- Log error messages and monitor exception handling behaviour.

## Process and Risk

Process

1. Early Development:
    - Develop and validate scaffolding (mock databases for restaurants and orders).
    - Begin testing R1 and R4 in parallel.
2. Mid-Development:
    - Conduct unit and integration testing for R2.
    - Use the validated mock databases for testing order validation and recording.
3. Late Development:
    - Test R3 using synthetic orders and validate outcome recording.

    - Perform final robustness tests and address any uncovered issues.

Risks

- Synthetic Data Limitations: Unrepresentative synthetic data may lead to untested edge cases and system failures. Mitigation: Carefully design test cases to cover a broad range of scenarios.
- Scaffolding Complexity: Building and maintaining mock databases could be resource-intensive. Mitigation: Allocate sufficient time early in the project and iteratively improve scaffolding.