LO2 Test Plan

1. Test Objectives

1.1 Verify Functional Completeness: Ensure the system accurately processes orders from reception through processing to drone delivery. This includes verifying all functionalities of the user interface, such as logging in, viewing restaurants and menus, placing orders, and the payment process, to ensure they work as expected.

1.2 Assess System Performance: Test the system's performance under various operational conditions, especially its stability under load. This involves testing response times to ensure the system responds to user requests within acceptable durations.

1.3 Robustness Verification: Validate the system's ability to handle exceptional inputs, such as incorrect or incomplete order information. The objective is to ensure the system can gracefully handle these situations, whether through appropriate error messaging or by reverting to a stable state.

1.4 Security and Privacy Protection: Confirm the system has implemented adequate security measures to protect user data against unauthorized access and data breaches. This includes testing access controls, data encryption, and privacy protection features of the system.

1.5 Usability Assessment: Test the system's user interface and overall user experience to ensure users can easily access and utilize system functionalities. This involves the clarity and timeliness of order status updates and guidance throughout the ordering and delivery process.

1.6 Delivery and Deployment Verification: Ensure the system can be successfully deployed in the intended environment, including compatibility testing with existing infrastructure. Additionally, validate the actual operation of the drone delivery process, ensuring drones can complete delivery tasks safely and accurately.

Through these test objectives, the ILP PizzaDronz project's testing activities will ensure that the software product meets not only functional requirements but also achieves high standards in performance, security, usability. This will help deliver a reliable and user-friendly drone pizza delivery service.

2. Design Test Strategy

2.1 Designing Test Suites Based on Testing Requirements:

Tailor test suites to address specific requirements outlined in section 1.2, breaking them down step by step to cover each requirement.
Consider whether separate test suites are needed for different levels of testing or if some can be shared. For instance, the same set of order data might be used to test both system-level functionalities like user ordering and drone delivery modules.

Integration Testing:

Design distinct test suites for integration testing focusing on the interaction between modules. This includes verifying the correct exchange of information between the order system, the restaurant system, and drone communication with the data center.

Unit-Level Testing:

Some functionalities require unique test suites, such as obstacle avoidance and pathfinding, which necessitate designing specific map data.
Other functionalities like menus and order data can share test suites, optimizing testing efforts and resources.

Performance Testing:

System performance under various conditions, including load stability, is a critical aspect of the test plan. Performance testing assesses the system's responsiveness and efficiency, ensuring it meets the set performance criteria.

2.2 Priority of Requirements

Highest Priority Requirements:
FindValidateOrders (Order Validation Functionality): This is one of the core functionalities that ensures all order information is accurate and meets processing criteria. Testing should ensure that all order scenarios are correctly identified, validated, and appropriately responded to for invalid or incomplete orders.

PathFindingAlgo (Path Finding Algorithm): For a drone delivery service, an efficient and accurate pathfinding algorithm is crucial. Testing needs to verify the algorithm's performance under different conditions and environments, ensuring drones can select the optimal route for delivery.

Secondary Priority Requirements:
System Performance: This includes response times and the system's stability under high load. While these requirements are vital for user experience, they support rather than directly determine the execution of core business processes.

2.3 Evaluation of the Quality of the Test Plan

Advantages: The test plan is rich in test suites, covering testing at all levels. Sharing some test cases across functionalities reduces testing workload while effectively evaluating system performance and stability.
Disadvantages: The scale of test data is limited, which may not fully test the system's performance under high user load conditions.

2.4 Instrumentation of the Code

Test Results Instrumentation
Purpose: To verify whether the test results meet the expected goals, including both functional and non-functional test outcomes.
Method: Inserting assertions or logging within the code can validate outputs or behaviors against expected outcomes during the execution of specific functionalities or test cases. This facilitates the automation of the testing process and the real-time capture of failed test scenarios.
Advantages: Quick identification and pinpointing of issues, enhancing testing efficiency and accuracy.

Performance Monitoring Instrumentation
Purpose: To monitor system performance, including response times, throughput, and resource utilization, to identify performance bottlenecks.
Method: By incorporating performance counters, timers, or monitoring tools, performance data can be collected during system runtime. This data aids in analyzing performance trends and identifying inefficient code segments.
Advantages: Helps optimize system performance, ensuring software stability under high load conditions.

Test Coverage Instrumentation
Purpose: To ensure that testing covers all critical paths and functionalities within the codebase.
Method: Utilizing test coverage tools to track which parts of the code are executed during testing. This includes function calls, conditional branches, and loops.
Advantages: Reveals areas of code that are not tested, guiding the creation and improvement of test cases, increasing code test coverage.

Security Analysis Instrumentation
Purpose: To identify security vulnerabilities and potential threats within the code, such as SQL injection, cross-site scripting (XSS), and unauthorized data access.
Method: By integrating security checks into the code, or employing static and dynamic analysis tools to assess security. These tools can automatically detect common security issues and provide recommendations for fixes.
Advantages: Early detection and remediation of security vulnerabilities, enhancing software security, and preventing future security threats.

Through these instrumentation techniques, the ILP PizzaDronz project can gain valuable insights during the development and testing phases, helping to improve the software's quality, performance, test coverage, and security. This ensures that the final product meets user needs and achieves success in the market.

The instrumentation for Test Results, Performance Monitoring, and Test Coverage is adequate and crucial for the testing process, providing valuable insights into the system's functionality,

performance, and coverage.
However, Security Analysis instrumentation requires more careful consideration. More rigorous security checks and vulnerability assessments should be integrated into the test strategy to ensure the system's security posture is robust and resilient against potential threats.

2.5 Testing Core Functionalities: OrderValidator

Testing the OrderValidator functionality could involve various scenarios, such as:

Invalid card number (CARD_NUMBER_INVALID)
Invalid expiry date (EXPIRY_DATE_INVALID)
Invalid CVV (CVV_INVALID)
Exceeding the maximum pizza count (MAX_PIZZA_COUNT_EXCEEDED)
Undefined pizza (PIZZA_NOT_DEFINED)
Incorrect total amount (TOTAL_INCORRECT)
Orders from multiple restaurants (PIZZA_FROM_MULTIPLE_RESTAURANTS)
Ordering from a closed restaurant (RESTAURANT_CLOSED)

These scenarios ensure that the OrderValidator accurately processes transactions under various conditions, contributing to the overall reliability and robustness of the ILP PizzaDronz system.

3 Test Environment

3.1 Planning the Test Environment

Identify Target Platforms: Determine the operating systems (e.g., Windows, macOS, Linux, Android) and architectures (32-bit or 64-bit) the project needs to support.

Setup Test Environments: Configure the necessary hardware, software, and network settings for each platform to closely simulate real-world scenarios.

Use Cross-Platform Tools: Select automation tools capable of running across different operating environments to enhance testing efficiency.

Leverage Virtualization: Utilize virtual machines and container technology for flexible creation and management of test environments.

Gray Box Testing: Employ Gray Box Testing methods, which combine understanding of the system's internals, to assess the impact of different operating systems on system performance.

3.2 Planning Test Data

Ensure Data Variety: Generate test data covering a wide range of user scenarios, including edge cases and anomalies.

Performance Test Data: Create data that simulates heavy user traffic to test system performance.

Security Test Data: Construct specific datasets to test the system's security defenses, such as against SQL injection and cross-site scripting attacks.

Data Management: Use tools to effectively manage test data, ensuring consistency and repeatability of tests.

Through these steps, a comprehensive testing environment can be established for the ILP PizzaDronz project, ensuring thorough testing under various conditions, thereby enhancing the final product's quality and user satisfaction.

4 Integrating testing activities into the selected software development lifecycle

4.1 Early Integration (Requirements and Design Phase)

Requirement Validation: Immediately after requirements are defined, initiate requirement validation tests to ensure all requirements are testable, clear, and meet stakeholders' expectations. This helps identify any ambiguities or unrealistic requirements early on.

Design Review and Testing: Incorporate static testing techniques during the design phase to review the system architecture and design documents. Use tools and checklists to ensure the design supports testability and aligns with defined requirements.

4.2 Continuous Integration (Development Phase)

Unit Testing: Integrate unit testing into the development phase, where developers write and run tests for individual units or components as they are developed. This can be facilitated by adopting a Test-Driven Development (TDD) approach, where tests are written before the actual code.

Integration Testing: As modules are developed and integrated, perform integration testing to ensure they work together as expected. This can be part of a Continuous Integration (CI) process, where code merges trigger automated integration tests.

4.3 Pre-Release Integration (Testing Phase)

System Testing: Conduct system testing once all components are integrated into the complete system. This tests the system as a whole to ensure it meets the specified requirements.

Performance and Security Testing: Integrate performance testing to evaluate the system's behavior under various conditions and loads. Similarly, conduct security testing to identify vulnerabilities and ensure the system's security measures are effective.

4.4 Post-Release Integration (Maintenance Phase)

Regression Testing: After the system is deployed, integrate regression testing into the maintenance phase to ensure new changes do not adversely affect existing functionalities. Automated regression

tests can be run as part of the deployment process.

User Acceptance Testing (UAT): Though typically conducted by end-users or stakeholders before final system acceptance, integrating UAT feedback into the maintenance phase can help quickly address any post-deployment issues or enhancements.