## *LO1.1:*

**Range of Requirements:**

For users to have full functionality over the system, a range of requirements must be met, such as receiving orders from a customer, validating orders, and autonomously operating the drone without exceeding the drones limitations such as only 2000 moves (each move is defined as moving 0.00015 degrees in any of the 16 cardinal compass directions, or maintaining position).

**Functional requirements:**

- User Login and Authentication

For a user to be able to create orders, they need to have an account so that we can see the orders linked to them in case of any issues they would like to report.

- Data Input and Validation

Orders placed have restrictions on them, max number of pizzas and only being from 1 restaurant, so orders need to be validated, as well as making sure the items ordered do exist.

- Data Storage and Retrieval

Restaurant coordinates, menu, remaining moves for the drone, and the coordinates of the food drop-off need to be stored somewhere and also later retrieved for flight calculations, order validation, etc. Although this is something that can be hard coded in a very small scale, for scalability purposes it should be done the proper way.

- Data Processing

The drone must be able to determine the most optimal route to conserve moves done to have the highest possible number of orders delivered.

- User Permissions and Access Control

As consumers have the ability to create orders, there has to be a way for restaurants to edit their menus. Different user permissions and access controls for Consumers therefore is a requirement.

- Correctness/Safety

If the drone goes over the allowed 2000 moves, it essentially runs out of charge and will crash. This is the worst-case scenario for many of the parties involved and we need to ensure this never happens. Most importantly, the general public, anyone who

was near the drone when it crashed could be injured. For the consumer, it means they never get their delivery. For investors, it results in loss of income as they would have to pay to replace the drone, and are not making any money until it is replaced, as well as likely liable for anyone who was injured by the drone.

### Measurable quality attributes

The most important measurable quality attribute this system relies on is performance.
The system should be efficient in calculating its route as it plays a role in delivery time and should be as low as possible to maintain customer satisfaction. Other measurable attributes would be:

Drone Movement Accuracy: How accurately does the drone follow the most efficient path calculated.
Customer satisfaction: The percentage of customers who rate their delivery experience as positive.
Delivery reliability: The percentage of deliveries completed as scheduled without any delays.
Food temperature: The average temperature of food delivered to customers, measured at the time of delivery.

### Qualitative requirements

User-friendly: The system should be easy to use and understand for customers.

Secure: The system should protect the personal and payment information of its users.

Efficient: The system should be able to handle large volumes of orders and deliveries with minimal delays.

Customizable: The system should allow users to personalize their experience, such as dietary restrictions.

User-centered: The system should prioritize the needs and satisfaction of its users.

## *LO1.2*

### System Requirements

These tests are required to test the entire system as a whole and if it functions as required. This would test the order placement, the delivery, and all relevant interactions.

### Integration Requirements

The system contains many working parts, that rely on each other for the system to function properly as a whole. This level of testing tests how the components interact with each other, and if they function as expected. This helps us ensure that the system sends and receives the correct information from the restaurants, such as informing the restaurant of the order placed, and the restaurant confirms that they are able to accept such order. The system also needs to be able to communicate with the drone and deliver it the correct flight path for the order the drone is supposed to fulfill.

### Unit Requirements

This level of testing tests each and every component of the system and ensures it functions as required and knows how to handles errors. This will ensure that the flight path calculation, the ordering system, payment system, and order validation system all individually work as intended.

## *LO1.3 & LO1.*

As this system is complex and contains multiple parts all working in tandem, many if not all of the testing approaches mentioned below would be combined with regression testing to ensure the changes do not have an unexpected impact on a different part of the system.

### Limit Testing

The first testing approach I would take is limit testing. The system as a whole fails if the drone takes over 2000 moves as it runs out of charge. This testing can be done with the unit testing, ensuring that the drone doesn't accept any orders that take more moves than it has remaining for a round-trip. It can also test invalid orders such 0 pizzas ordered or more than 3 pizzas in one order. The reason I would go for this approach first is because these edge cases often represent a higher risk of error, as they can cause unexpected behavior or produce incorrect results. Limit testing can be performed at various levels of testing and is useful for verifying the system's robustness and stability.

### Black-Box Testing

Black-Box testing is based on the specification and requirements of the system and tests the system's response to various inputs and scenarios. The goal of this testing is to verify that the system behaves as expected and meets the requirements, by testing its inputs, outputs, and interactions with other systems. This approach is useful for identifying issues such as incorrect input validation, missing or incorrect functionalities, and incorrect handling of error conditions.

## White-Box Testing

The goal of structural testing is to ensure that the code is correct, complete, and free of defects, by verifying that all possible paths through the code are executed and that all branches and loops are tested. This approach can help identify issues such as logic errors, unhandled exceptions, and missing or incorrect implementation of algorithms.