Testing Requirements

1. **System-level requirements**
   a) **The system must ensure drones never enter any no-fly zones at any time**
      i. **Level**: Since this requirement constrains the observable behaviour of the whole system, which requires the cooperation of Model, View and Controller, this is a system-level requirement.
      ii. **Type**: Functional requirement
      iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)
      iv. **Appropriateness**:
         1. This requirement is naturally expressed in terms of input category partitions, such as the number and positions of the no-fly zones, and the relative positions of the start and delivery points. Category-partition testing explicitly asks us to identify and define such category partitions.
         2. From those partitions, we can derive a small but systematic set of test scenarios. For example: a map with no no-fly zones, a map where no-fly zones do not affect the route, and a map where no-fly zones force a detour. In summary, each of these test scenarios lets us verify that the algorithm ensures the drones never enter a no-fly zone while still finding a valid route when one exists.
         3. Alternative combinatorial approaches are less suitable here. Pairwise testing is useful when we have many independent configuration parameters and want to cover all two-way interactions between them. For this requirement, the main complexity lies in the positions and extent of the no-fly zones, so there aren't many combinations (pairs) to test. Catalogue-based testing would be attractive if we had an established catalogue of typical no-fly zone patterns from past projects, but in this project, building such a catalogue from scratch would be too much effort compared to defining partitions directly.
   b) **The system must ensure that drones do not fall from the sky during the delivery**
      i. **Level**: Since this requirement constrains the observable behaviour of the whole system, which requires the cooperation of Model, View and Controller, this is a system-level requirement.
      ii. **Type**: Functional requirement
      iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)

iv.

v. **Appropriateness**:

1. This requirement comes with partitions. For example, we can construct our partitions based on the relative position and number of orders: there are not too many orders and any drones can be picked; there are not many orders, but they are far from each other, which means either multiple deliveries must be made or a powerful drone is required; there are too many orders which means multiple deliveries are always required
2. Pairwise testing is not suitable here since there aren't many configuration parameters. Catalogue-based testing is also not a good option for the reason illustrated earlier.

c) **When making deliveries and returning to base, the distance between the drones and the target must not be greater than 0.00015 radius**

i. **Level**: Since this requirement constrains the observable behaviour of the whole system, which requires the cooperation of Model, View and Controller, this is a system-level requirement.

ii. **Type**: Functional requirement

iii. **Testing approach**: Black-box testing

iv. **Appropriateness**

1. This requirement does not depend on any configuration variable and does not come with any obvious partitions. A simple black-box test that examines when points where the drone is making deliveries or returning to base are close enough to the corresponding required delivery point or drone base should be sufficient.

d) **Drones must hover at every delivery point**

i. **Level**: Since this requirement constrains the observable behaviour of the whole system, which requires the cooperation of Model, View and Controller, this is a system-level requirement.

ii. **Type**: Functional requirement

iii. **Testing approach**: Black-box testing

iv. **Appropriateness**

1. This requirement does not depend on any configuration variable and does not come with any obvious partitions. A simple black-box test that examines whether a coordinate that is close enough to a delivery point appears twice (which indicates a hover) should do the job

e) **Drones must return to their base after all deliveries are made**

i. **Level**: Since this requirement constrains the observable behaviour of the whole system, which requires the cooperation of Model, View and Controller, this is a system-level requirement.
ii. **Type**: Functional requirement
iii. **Testing approach**: Black-box testing
iv. **Appropriateness**
    1. This requirement does not depend on any configuration variable and does not come with any obvious partitions. A simple black-box test that examines whether there is always a sub-path in every path that has an origin of the drone's last hover point and a destination of a position near the drone's base should be sufficient

f) Drones must not make multiple deliveries on different dates in one go

i. **Level**: Since this requirement constrains the observable behaviour of the whole system, which requires the cooperation of Model, View and Controller, this is a system-level requirement.
ii. **Type**: Functional requirement
iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)
iv. **Appropriateness**:
    1. This requirement is expressed in two simple partitions: all orders are the same date, or orders are of different dates. Category-partition testing explicitly asks us to identify and define such category partitions. Then, from those partitions, we can easily derive the two corresponding test cases.
    2. Pairwise testing adds overhead because there is only one input configuration parameter, which yields the two derived test scenarios above. For the same reasons outlined above, catalogue-based testing is not appropriate here.

g) The system must ensure that the average cost per delivery along a path is less than the maximum cost of any deliveries made on that path, if the maximum cost is provided

i. **Level**: Since this requirement constrains the observable behaviour of the whole system, which requires the cooperation of Model, View and Controller, this is a system-level requirement.
ii. **Type**: Functional requirement
iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)
iv. **Appropriateness**

1. Category-partition testing is still the best option here, as we can construct the following partitions: orders are close to each other, and there are efficient drones available (which can deliver all orders and not exceed the cost), orders are close to each order but there are no efficient drones available which means multiple delivery paths are required, orders are far from each other which means regardless of the drones multiple routes are needed.
2. Pairwise testing is not appropriate because the test cases are based on relative positions, the number of orders, and the available drones. Catalogue-based testing is not applicable, as illustrated before.

h) The system must ensure that the drone is not overloaded when planning a path with some deliveries for that drone

i. **Level**: Since this requirement constrains the observable behaviour of the whole system, which requires the cooperation of Model, View and Controller, this is a system-level requirement.
ii. **Type**: Functional requirement
iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)
iv. **Appropriateness**:
    1. This requirement can be easily partitioned based on the number of orders and the relative positions of the orders and the drones available. For example, we can have: there are not many orders and a drone which has a large enough capacity to make all deliveries in one go; there are not many orders, but the drone does not have a large enough capacity, which means multiple deliveries are needed; there are too many orders and multiple deliveries are required regardless of the drones available.
    2. Pairwise testing is not suitable here since there aren't many configuration inputs and the partitions are based on relative positions, the number of orders and the drones available. Catalogue-based testing is not a good option for the reasons outlined.

i) The system must ensure the drone used for one path is available for all deliveries along that path

i. **Level**: Since this requirement constrains the observable behaviour of the whole system, which requires the cooperation of Model, View and Controller, this is a system-level requirement.
ii. **Type**: Functional requirement
iii. **Testing approach**: Functional category-partition testing (black box,

combinatorial)

iv. **Appropriateness**:

1. This requirement can also be partitioned based on the delivery time for each order. For example, we can have the following partitions: all orders have similar delivery times, so finding a suitable drone for all of them is easy; the delivery times vary widely, so multiple drones must be used.
2. Pairwise testing adds overhead here, as there is only one configuration variable. Catalogue-based testing is not a good option for the reasons mentioned before.

j) The system must ensure that the selected drone meets the heating and cooling requirements for deliveries along one path.

i. **Level**: Since this requirement constrains the observable behaviour of the whole system, which requires the cooperation of Model, View and Controller, this is a system-level requirement.

ii. **Type**: Functional requirement

iii. **Testing approach**: Functional pairwise testing (black box, combinatorial)

iv. **Appropriateness**

1. Pairwise testing is the best option here, as it allows constructing a relatively small number of test cases based on the configuration parameters. For each order, cooling and/or heating may be required. So, naturally, we can use a binary pair of those two configuration parameters (heating, cooling) to construct our test scenarios, yielding four potential scenarios.
2. Category-partition testing may also be used here, as the partitions are relatively clear, but using pairs makes the test cases more explicit. Catalogue-based testing is not appropriate, as illustrated before.

k) The system must return the calculated delivery path in three minutes

i. **Level**: Since this requirement constrains the observable performance of the whole system, which requires the cooperation of Model, View and Controller, this is a system-level requirement.

ii. **Type**: Since the system's response time can be measured using simple timing, it is a measurable quality attribute.

iii. **Testing approach**: System testing/Stress testing

iv. **Appropriateness**:

1. System testing is part of the system verification, especially for global properties such as performance and reliability. The system's response time is clearly part of its performance, which can be verified through

system testing.

2. Stress testing is a specific kind of system testing which involves an expensive simulation of the execution. For example, add 100 orders to the system at once and check whether the system still responds within 3 minutes. Although stress testing provides confidence in typical load envelopes, it cannot guarantee performance in every possible environment configuration.

l) **The system should always return a 200 code**

i. **Level**: Since this requirement constrains the observable behaviour of the whole system, which requires the cooperation of Model, View and Controller, this is a system-level requirement.

ii. **Type**: Functional requirement

iii. **Testing approach**: System testing

iv. **Appropriateness**:

1. Again, since this is a property of the system which can be observed by inspecting the status code of the system response, system testing should be used here.

m) **The total cost/move of a delivery path calculated by the system should be within ±5% of the real or optimal cost/move for the delivery.**

i. **Level**: Since this requirement constrains the observable behaviour of the whole system, which requires the cooperation of Model, View and Controller, this is a system-level requirement.

ii. **Type**: measurable quality attribute

iii. **Testing approach**: System testing

iv. **Appropriateness**:

1. System testing is part of the system verification, especially for global properties such as performance and reliability. The total cost/moves of the path calculated by the system is part of its performance properties, since we want the system to deliver using the fewest possible moves/cost.

2. **Integration-level requirements**

a) **When the calcDeliveryPathAsGeoJson endpoint is called, the system must retain the calculated delivery path coordinates when returning the path in a valid GeoJSON format**

i. **Level**: This requirement constrains how the code of the calcDeliveryPathAsGeoJson endpoint interacts with the code of the

calcDeliveryPath endpoint. This is an integration-level requirement that governs the interaction between two pieces of code.

ii. **Type**: Functional requirement

iii. **Testing approach**: Black-box testing

iv. **Appropriateness**:

1. This requirement is about ensuring the coordinates are not changed when remapping the result from the calculateDeliveryPath endpoint to another format. So, a simple black-box test that verifies the coordinates are the same using the results from the calcDeliveryPath and calcDeliveryPathAsGeoJson endpoints, both based on the same input, is good enough.

b) When the calcDeliveryPath endpoint is called, the system must request the necessary data every time an endpoint is called from the URL indicated by an environment variable or from the default URL

i. **Level**: This requirement constrains how the get data service interacts with the system configuration, which is clearly an integration-level requirement

ii. **Type**: Functional requirement

iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)

iv. **Appropriateness**:

1. This requirement comes with at least two partitions: URL specified by the environment variable, and URL not specified by the environment variable. So, a simple category-partition testing with the two above test scenarios can be used
2. Pairwise testing is not needed here because there is only one configuration variable. Catalogue-based testing is also not ideal, as illustrated.

c) When the calcDeliveryPath endpoint is called, the system must separate the list of given orders by date and time

i. **Level**: This requirement is about how the path planning function interacts with other utility functions to correctly process the raw data, which is at the integration level

ii. **Type**: Functional requirement

iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)

iv. **Appropriateness**:

1. We can have two kinds of input partitions: all orders are on the same date, or orders are on different dates. Then we can check if the

separated sublists only contain the order of the same data and are ordered by time. So again, category-partition testing is ideal

2. Pairwise is not suitable due to the lack of configuration variables, and the catalogue-based testing is also not ideal due to the lack of a catalogue

d) **The system must correctly calculate the move based on the returned path from the aStarSearch function**

i. **Level**: This requirement puts constraints on how the function for planning the path interprets the result from the function aStarSearch, which is again, at the integration level

ii. **Type**: Functional requirement

iii. **Testing approach**: Black-box testing

iv. **Appropriateness**:

1. Based on the ILP document, a drone move corresponds to moving from one coordinate to another, even if the two coordinates are the same. So, a black-box testing which verifies the hover point is correctly added to the coordinate list returned from the aStarSearch, and the move is equal to the length of that list minus one, is desired.

e) **The system must validate that each drone has a valid service-point mapping: a drone ID must appear in at least one servicePointDrones entry, and the corresponding servicePointId must exist in the servicePoints list**

i. **Level:** This requirement limits how the path planning function uses other functions to extract and check whether a drone has a valid availability mapping, which is also at the integration level.

ii. **Type**: Functional requirement

iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)

iv. **Appropriateness**

1. Again, we have two partitions: the drone has a valid service point, and the drone does not. We should test that in the first scenario, the system accepts the drone and continues with the planning, and for the second one, the system should simply reject the drone and move on. So, category-partition testing can be used

2. There are no parameters to use or pass in, so pairwise testing does not fit. Catalogue-based testing is not suitable, as illustrated.

f) **The system must build the return structure correctly such that every delivery and the returning paths of a drone are separated, any hovers are**

indicated, and drone IDs organise delivery paths

i. **Level**: This is about how the path planning function calls the other function for building the data structure used to return data, which is at the integration level
ii. **Type**: Functional requirement
iii. **Testing approach**: Black-box testing
iv. **Appropriateness**
    1. This requirement is about checking that the returned data structure is not malformed, which can be tested with black-box testing and some scaffolding code that records the intermediate results produced during path planning.

g) The system must check if the next move results in the drone entering any no-fly zones or crossing any edges of any no-fly zones

i. **Level**: The aStarSearch function must call the functions for checking that the next move is not cross any edge of any no-fly zones, and it is not inside any no-fly zones, so still the above requirement is at the integration level
ii. **Type**: Functional requirement
iii. **Testing approach**: Functional category-partition testing / pairwise testing (black box, combinatorial)
iv. **Appropriateness**
    1. We've got two parameters here: next move crossing any edge of any no-fly zones, and subsequent move is inside any no-fly zones. So, using them, we have four partitions: both the first and second parameters are true, only the first is true, only the second is true, and neither is true. So, we can either use pairwise (1 pair) testing or category-partition (4 partitions) testing.
    2. Catalogue-based testing is not considered.

h) The system must ensure that any point sufficiently close to any previously visited point is never going to be visited

i. **Level**: This is about how the aStarSearch function uses the hashset data structure and another function which produces the hash value interacts to prevent the algorithm from getting trapped in one area forever, which is at the integration level
ii. **Type**: Functional requirement
iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)
iv. **Appropriateness**

1. Two observable partitions: the next chosen move is close to a visited point, and the next chosen move is not close to any visited point. A simple category-partition testing, which checks the system should reject the first scenario and accept the second scenario, will get the job done
2. No input parameter here, so pairwise testing cannot be used. Catalogue-based testing is not considered.

3. **Unit-level requirements**
   a) **The getAllDates function should correctly identify all unique dates from a list of given orders**
      i. **Level**: This requirement governs the functionality of a single function, which means it is at the unit level
      ii. **Type**: Functional requirement
      iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)
      iv. **Appropriateness:**
         1. We can potentially make three partitions: the input is empty, the input contains only orders of the same date, or the input contains orders of different dates. As a result, category-partition testing can be used.
         2. Although the list of input orders can be viewed as a parameter, since there is only one parameter, pairwise testing adds overhead. Catalogue-based testing is not considered.
   b) **The getMedicineDispatchByDate function should correctly extract the orders of a specific date from a list of given orders**
      i. **Level**: This requirement governs the functionality of a single function, which means it is at the unit level
      ii. **Type**: Functional requirement
      iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)
      iv. **Appropriateness:**
         1. We can, for example, have the following partitions: the input date is empty, there are some orders of the input date, and there are no orders of the input date. So, category-partition testing may be used.
         2. Pairwise testing adds overhead, and catalogue-based testing is not considered

c) **The getServicePointPositions function should correctly return the service point position of a drone or Null given the drone's ID, a list of service points, and a date and time.**
   i. **Level**: This requirement governs the functionality of a single function, which means it is at the unit level
   ii. **Type**: Functional requirement
   iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)
   iv. **Appropriateness:**
      1. The output of this function depends on four parameters: drone ID, list of service points, date and time. This means we can construct partitions, such as whether the drone is available at one service point at the given date and time and whether it is not available at any service point at that time.
      2. On the other hand, it would be impossible to consider all possible inputs for those four parameters, and even if we limit ourselves to pairwise combinations, we would still have too many test scenarios, so pairwise testing is not suitable. Catalogue-based testing is not considered

d) **The segmentIntersects function should correctly determine whether two segments intersect or not based on the calculated orients**
   i. **Level**: This requirement governs the functionality of a single function, which means it is at the unit level
   ii. **Type**: Functional requirement
   iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)
   iv. **Appropriateness**:
      1. It is easy to construct partitions. We may have: the two segments intersect at a non-vertex point, the two segments do not intersect, the two segments intersect at a vertex but not colinear, the two segments intersect at a vertex and colinear, or the two segments are colinear but do not intersect. So it is natural to use category-partition testing.
      2. There are no obvious configuration parameters, so pairwise testing is not suitable. Catalogue-based testing is not considered

e) **The isInRegion function should confirm a position is inside a no-fly zone if the horizontal ray with that position as its vertex has an odd number of**

intersections with the no-fly zone edges

i. **Level**: This requirement governs the functionality of a single function, which means it is at the unit level
ii. **Type**: Functional requirement
iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)
iv. **Appropriateness**:
    1. We can have input partitions. A good example will be: the point to be tested is outside the region, the point is inside the region, the point is on an edge of the region, the point is on a vertex of the region. So category-partition testing is preferred
    2. Again, there would be too many test cases even if we only consider the pair combination of the possible input values. Catalogue-based testing is not considered

f) **The aStarSearch function should stop searching for a path after a certain number of loops**

i. **Level**: This requirement governs the functionality of a single function, which means it is at the unit level
ii. **Type**: Functional requirement
iii. **Testing approach**: Functional category-partition testing (black box, combinatorial)
iv. **Appropriateness**:
    1. Two partitions: the goal is not far from the origin, and the goal is far from the origin, so category-partition testing can be used.
    2. Pairwise-testing adds overhead and catalogue-based testing not considered