Testing requirements

- R1: The system should be able to validate orders accurately
- R2: The system should find the most optimal path without entering no-fly zones
- R3: The system should adhere to the 60 second runtime constraint

R1:

To test this requirement I used the systematic functional approach by conducting unit tests for key components, including credit card information, order details and restaurant information. This was to make sure that the code met the functional requirements of the system. I specifically tested for edge cases for valid and invalid inputs. I also made sure to test not only the edge cases but also boundary values (just inside, on, and outside defined boundaries) with boundary value analysis testing to make sure the systems robustness. I further conducted system level tests to make sure that entire validation pipeline flags invalid orders appropriately and to verify that only valid orders proceed to downstream processes like pathfinding and file generation. At the same time, valid orders were verified to proceed seamlessly through the validation process. This comprehensive approach aimed to guarantee the accuracy and reliability of the order validation system under a wide range of conditions.

Test cases

Test type	Input/condition	Expected outcome
Credit card validation	Valid CVV (3 digits)	Accepted
	Invalid CVV (2 or 4 digits)	Rejected with appropriate validation code
	Valid card number (16 digits)	Accepted
	Invalid card number (15 or 17 digits, non-numeric)	Rejected with appropriate validation code
	Expired card date	Rejected with appropriate validation code
	Valid card expiry date	Accepted
Order Validation	0 pizzas	Rejected with appropriate validation code
	1 -4 pizzas	Accepted
	5 pizzas	Rejected with appropriate validation code
	undefined pizzas in order	Rejected with appropriate validation code
	Total matches price of pizzas plus order charge	Accepted
	Total differs by 1 pence	Rejected with appropriate validation code

	No pizza name	Rejected with appropriate validation code
	No pizza price	Rejected with appropriate validation code
Restaurant validation	Not existing pizza name in restaurant	Rejected with appropriate validation code
	All pizzas from one open restaurant	Accepted
	Multiple restaurants in order	Rejected with appropriate validation code
	Pizza price differs from order	Rejected with appropriate validation code
	Restaurant closed on the order day	Rejected with appropriate validation code
	No restaurants defined	Rejected with appropriate validation code
System level testing	Valid orders	Processed correctly → valid output files created
	Invalid orders - Singular validation failure	Rejected and excluded from further processes
	Invalid orders - Multiple simultaneous validation failures	Rejected and excluded from further processes

Evaluation/weaknesses:

The testing I conducted for R1 was comprehensive in many areas, including detailed unit testing, boundary value analysis, and system-level testing. I thoroughly tested individual components, such as credit card validation, pizza count validation, and restaurant validation, making sure that a wide variety of conditions were covered. I also verified that invalid orders, whether due to singular or multiple failures, were correctly excluded from further processes. However, I didn't focus as much on integration testing to ensure smooth data flow between components, such as validating how credit card details, order pricing, and restaurant data interact within a single order. If I had more time, I would prioritise integration testing to validate data consistency across components and confirm that all validation logic works cohesively.

R2:

I used category partitioning, a systematic approach, for figuring out test cases to use to test this requirement. This technique helped me to reduce the number of test cases while still maintaining broad coverage by focusing on the most important combinations. I then used a model based approach, by doing simulation test on the discovered test cases to generate different scenarios of no-fly zones to test the robustness of the system with synthetic data.

Categories

Origin:

- valid - within the delivery region
- invalid - outside the delivery region

Destination:

- valid: reachable destination
- invalid: destination blocked by no-fly zones

No-fly zones:

- None
- A few, easy pathfinding
- Dense, complex pathfinding

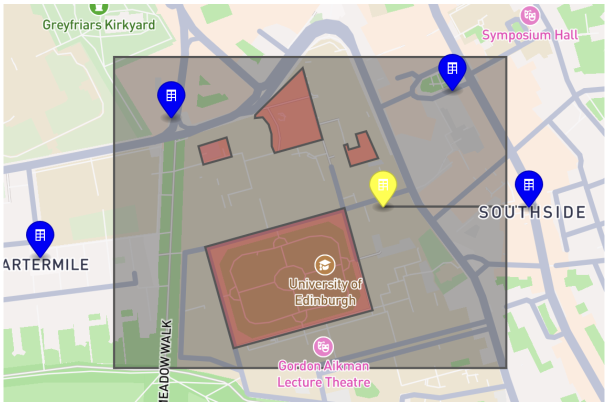
  Specific scenarios:

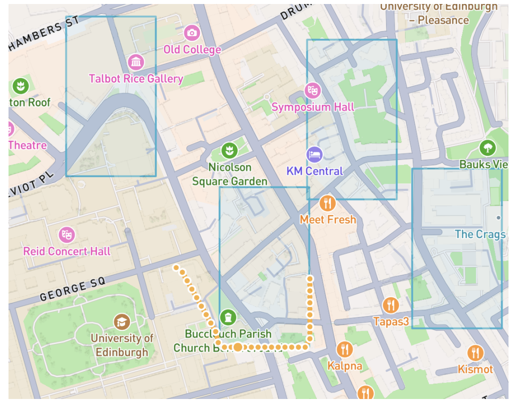
  - no fly zones intersect (can be removed as that is just a larger no-fly zone)
  - no fly zones covering the origin/destination (can be removed as that is not a real world scenario case and the algorithm doesn't need to work for such instance)
  - weirdly shaped no fly zones

From the above we can gather the following test cases:

- Test paths where origin or destination is invalid
- Simulate various no-fly zone densities and ensure paths avoid obstacles

However, the LngLatHandler class already implements validation for checking if points (origin or destination) are within a region or close to another point. Since these functionalities are thoroughly implemented and would already be indirectly validated during pathfinding and no-fly zone avoidance tests, explicitly testing origin and destination validity is redundant. This means that I can instead focus on the correctness of pathfinding around different no-fly zones scenarios.

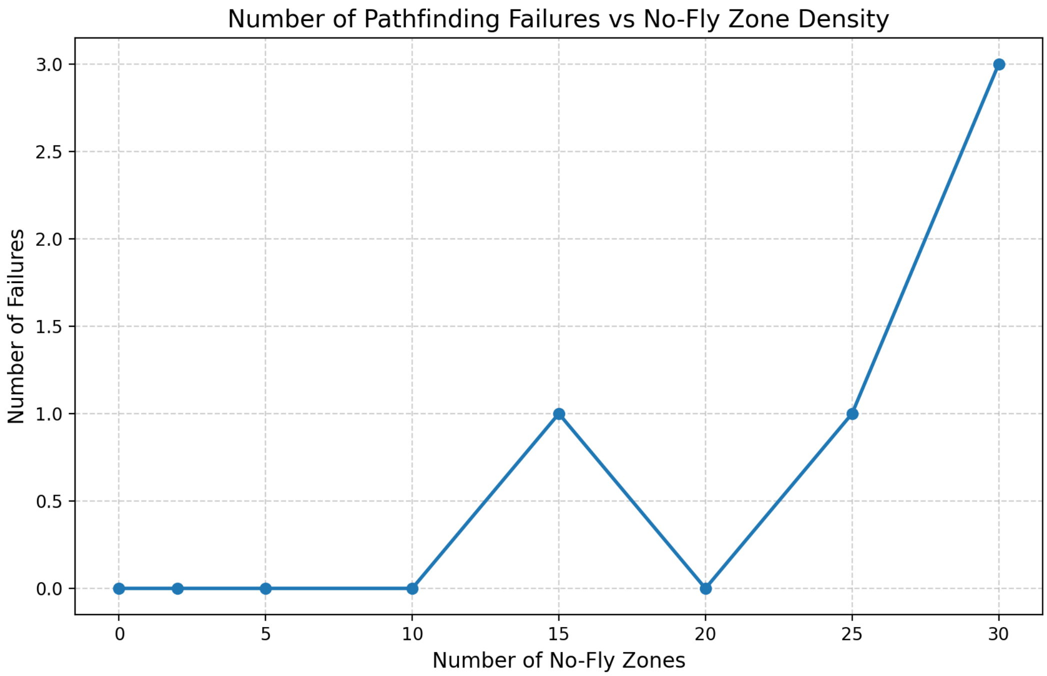




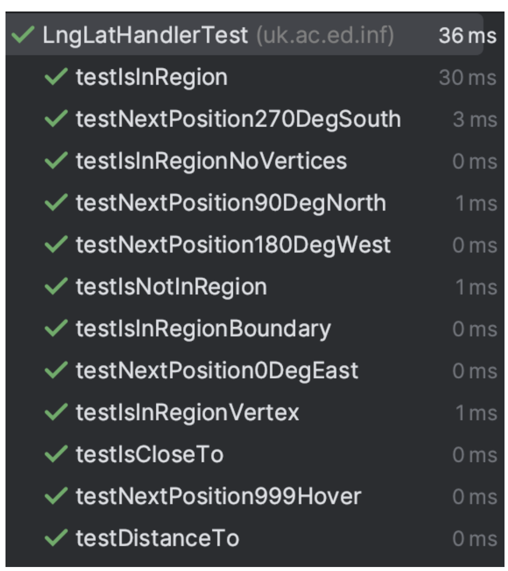
Number of Pathfinding Failures vs No-Fly Zone Density

- **Purpose:** To analyse how often the algorithm fails to generate a valid path as the density of no-fly zones increases
- **Metric:** Count the number of failures for different densities of no-fly zones

Number of No-Fly Zones	Test Run 1 (Success/Fail)	Test Run 2 (Success/Fail)	Test Run 3 (Success/Fail)	Test Run 4 (Success/Fail)	Test Run 5 (Success/Fail)
0	Success	Success	Success	Success	Success
2	Success	Success	Success	Success	Success
5	Success	Success	Success	Success	Success
10	Success	Success	Success	Success	Success
15	Success	Fail	Success	Success	Success
20	Success	Success	Success	Success	Success
25	Success	Success	Success	Success	Fail
30	Success	Fail	Success	Fail	Fail



I further implemented unit tests to validate the functionality of LngLatHandler, to have accurate calculations required for the pathfinding algorithm. These tests focused on verifying that points were correctly identified as being within or outside no-fly zones and that distances and movement constraints were calculated precisely. This was essential for the pathfinding algorithm so it could navigate efficiently and avoid errors when interacting with complex or dense configurations of no-fly zones



Evaluation/weaknesses:

I successfully tested standard scenarios, including realistic and no-obstacle configurations, which helped confirm the system's basic functionality. Additionally, I tested high-density no-fly zones, which provided insights into the system's behaviour under more complex configurations. I also addressed edge cases, such as paths grazing no-fly zone boundaries, though coverage of such tests might have been limited. Some focus was placed on validating specific components like LngLatHandler, making sure the core functionalities such as distance calculations and point identification within zones were accurate.

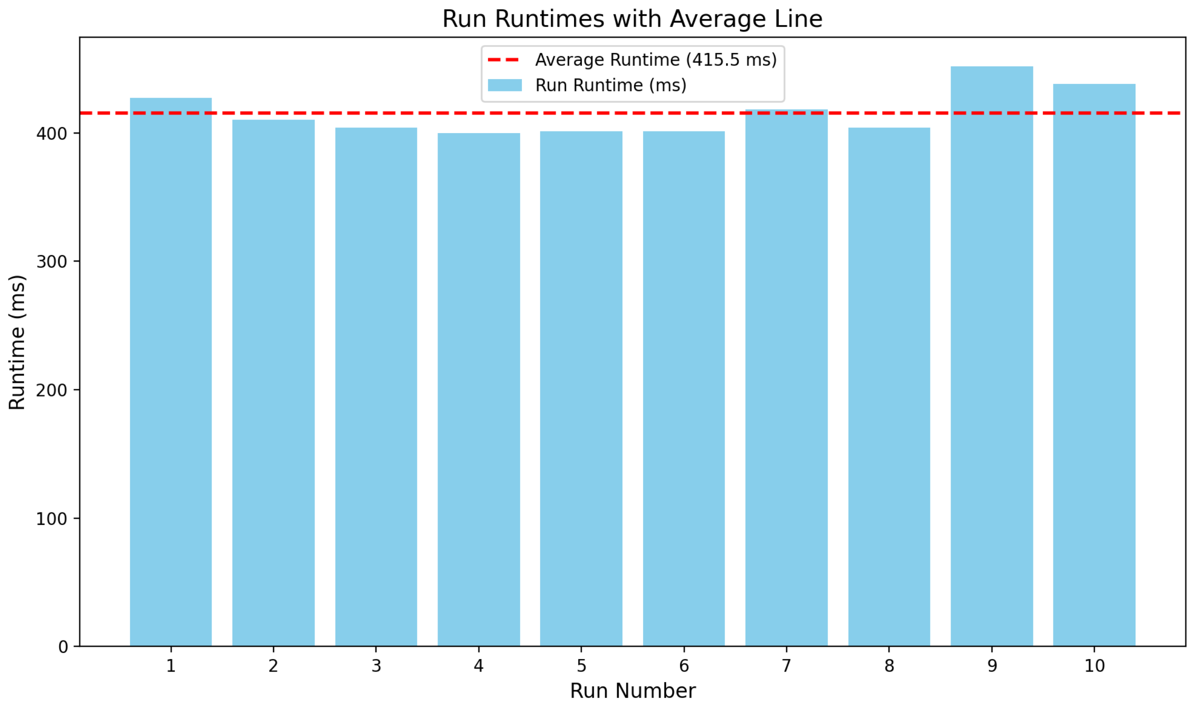
If I had more time, I would do stress testing by making a function that would create not only rectangular no fly zones but also more complicated shapes to test the durability better and test with 100+ no fly zones. Fuzz Testing was also not implemented as planned, given the project's scope and time limitations because generating and handling unpredictable inputs wasn’t a priority. I would further, expand unit tests to include additional methods and edge scenarios for LngLatHandler and other critical components, improving the overall reliability of the system.

R3:

To test this requirement, I made use of performance testing to test that the optimal path was found in the 60 seconds runtime constraint. To do this, I used 2 methods. Firstly, I simulated a scenario using synthetic data and tested the pathfinding algorithm on it own to test that that finds the optimal path in the required time. Next, I tested the runtime with the inclusion of the REST API service to test that the overall runtime was also within the required constraints for a large range of days.

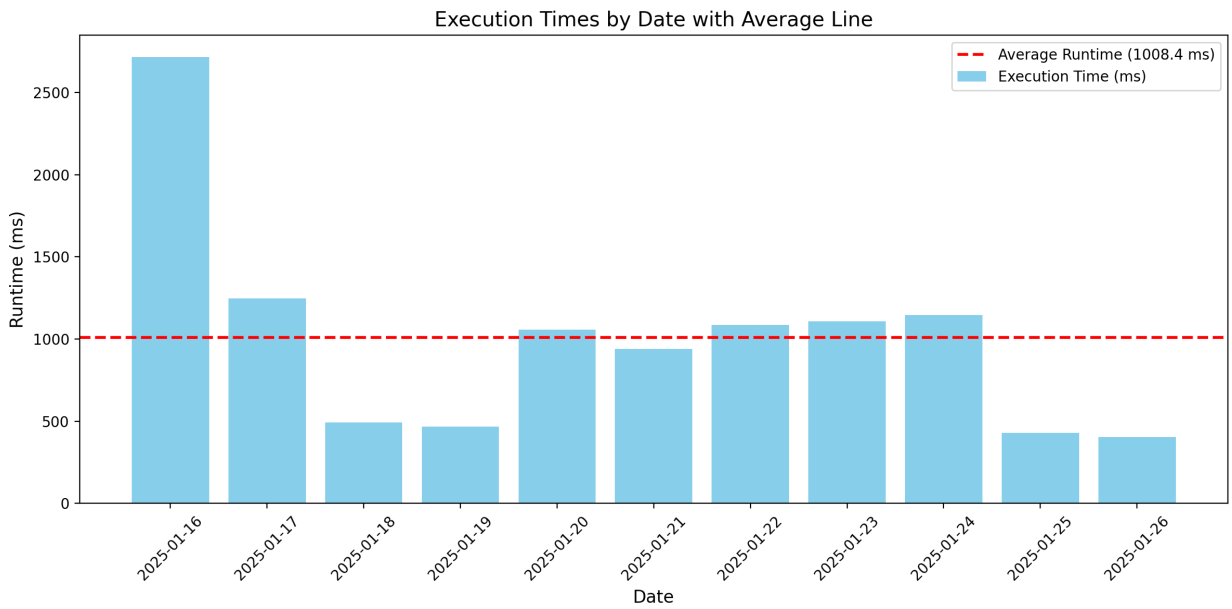
My own synthetic data

Run	**Runtime (ms)**
1	427
2	410
3	404
4	400
5	401
6	401
7	418
8	404
9	452
10	438
Average	**415.5**

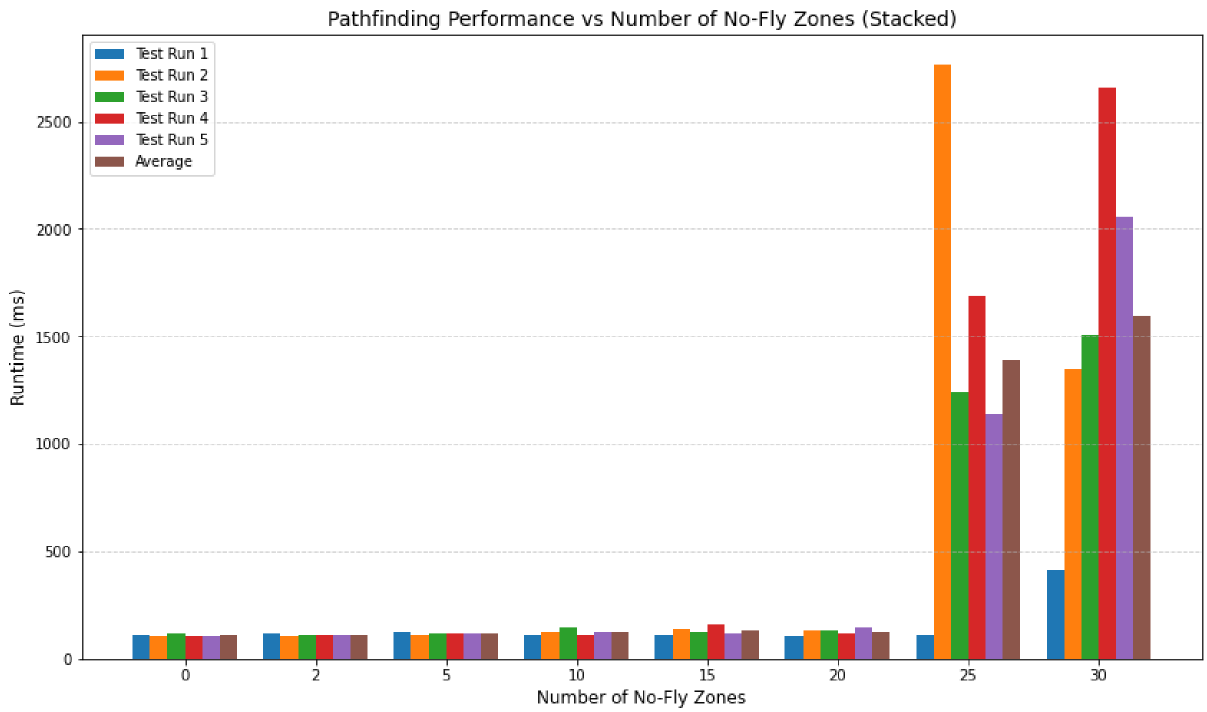


Using REST API

Run	Runtime (ms)
2025-01-16	2714
2025-01-17	1246
2025-01-18	493
2025-01-19	467
2025-01-20	1058
2025-01-21	940
2025-01-22	1086
2025-01-23	1107
2025-01-24	1146
2025-01-25	430
2025-01-26	405
Average	**1008.4**



Number of No-Fly Zones	Test Run 1	Test Run 2	Test Run 3	Test Run 4	Test Run 5	Average (ms)
0	109 ms	108 ms	121 ms	105ms	108 ms	110.2
2	118 ms	108 ms	110 ms	109 ms	113 ms	111.6
5	123 ms	115 ms	117 ms	116 ms	116 ms	117.4
10	112 ms	124 ms	145 ms	110 ms	125 ms	123.2
15	112 ms	139 ms	125 ms	159 ms	122 ms	131.4
20	105 ms	129 ms	131 ms	120 ms	146 ms	126.2
25	109 ms	2767 ms	1241 ms	1692 ms	1138 ms	1389.4
30	415 ms	1345 ms	1508 ms	2658 ms	2056 ms	1596.4



Evaluation/weaknesses:

I successfully tested the performance of the pathfinding algorithm to stick to the 60-second runtime constraint. I conducted performance testing under standard conditions with a typical number of no-fly zones, as well as high-stress scenarios with larger and more complex no-fly zone configurations. This included testing both the isolated algorithm and its integration with the REST API service to confirm the overall system's runtime stayed within the required limits. I also recorded and analysed runtimes, identifying trends in performance and making sure valid paths were generated consistently. However, I mainly focused on standard and high-density scenarios, with limited exploration of irregular configurations.

If I had more time, I would extend integration testing to assess how well the pathfinding algorithm interacts with the validation and output generation processes under many different input conditions. This would help identify potential delays or inefficiencies in end-to-end workflows. Finally, I would analyse path correctness and efficiency in greater detail, making sure the system balances runtime performance with the generation of optimal paths.

Coverage Testing

Lastly, I made sure to test with coverage, a structural testing approach to verify that the critical components of the system were thoroughly exercised during testing. Coverage testing allowed me to identify which parts of the code were executed by my tests and which parts were left untested, helping to focus efforts on areas that were most relevant to the three main requirements. This approach ensured that critical workflows and edge cases were addressed, minimising the risk of undetected issues.

I focused my efforts on achieving high coverage for the components and workflows directly tied to the three requirements. While I achieved near-complete coverage for R1 and R2’s critical components, some supporting classes remained untested. For R3, while runtime performance was validated, testing didn't extend to all edge cases or advanced geographic scenarios. If more time were available, I would expand coverage to include these untested components and scenarios to provide a more comprehensive evaluation.

Element	Class, %	Method, %	Line, %
uk.ac.ed.inf	86% (19/22)	81% (65/80)	69% (309/446)
App	0% (0/1)	0% (0/2)	0% (0/14)
AStarFlightPathSolver	100% (1/1)	100% (4/4)	90% (49/54)
CoordinatePoint	100% (1/1)	50% (2/4)	28% (7/25)
CreditCardInformationCheck	100% (1/1)	100% (1/1)	75% (21/28)
Delivery	100% (1/1)	100% (1/1)	100% (5/5)
Direction	100% (1/1)	100% (4/4)	100% (7/7)
FlightMove	100% (1/1)	100% (4/4)	100% (16/16)
FlightPath	100% (1/1)	100% (3/3)	100% (15/15)
GeoJSONFeature	100% (1/1)	100% (3/3)	100% (4/4)
GeoJSONFeatureCollection	100% (1/1)	100% (3/3)	100% (4/4)
GeoJSONGeometry	100% (1/1)	100% (2/2)	100% (3/3)
GetClient	0% (0/1)	0% (0/7)	0% (0/63)
Hover	100% (1/1)	100% (4/4)	100% (5/5)
IValidator	100% (0/0)	100% (0/0)	100% (0/0)
JsonFileCreator	100% (1/1)	100% (5/5)	97% (40/41)
LngLatHandler	100% (1/1)	100% (6/6)	96% (24/25)
OrderManager	0% (0/1)	0% (0/2)	0% (0/19)
Orders	100% (1/1)	88% (16/18)	76% (30/39)
OrdersStatus	100% (1/1)	100% (2/2)	100% (6/6)
OrdersValidation	100% (0/0)	100% (0/0)	100% (0/0)
OrderValidationCodes	100% (1/1)	100% (2/2)	100% (13/13)
OrderValidator	100% (1/1)	100% (1/1)	100% (13/13)
PizzaInformationCheck	100% (1/1)	100% (1/1)	100% (19/19)
RestaurantInformationCheck	100% (1/1)	100% (1/1)	100% (28/28)

Requirements Linked to Coverage Testing

R1: The System Should Be Able to Validate Orders Accurately:

- Makes sure that validation logic (credit card, pizza, and restaurant information) is thoroughly tested.
- Confirms that edge cases, boundary conditions, and integration of validation components are covered.

R2: The System Should Find the Most Optimal Path Without Entering No-Fly Zones:

- Validates the pathfinding algorithm under various scenarios.
- Confirms that logic handling no-fly zones and boundary constraints is well covered.

R3: The System Should Adhere to the 60-Second Runtime Constraint:

- Makes sure performance-critical sections of the code are tested.
- Confirms no redundant or inefficient lines exist, improving runtime efficiency.