Requirements

R1: The system must evaluate the validity of each order.

R2: The system must calculate the day's flightpath within 60 seconds.

R3: The system must build no-fly-zones using data from REST server.

**some very brief extras in the '*Extras*' document of the repository

Prioritising and pre-requisites

R1: This is a unit-level requirement made up of multiple sub-requirements, each dealing with a specific aspect of the order validation which must all be verified before **R1** can be verified.

By the partition principle, verifying **R1** will be much easier by decomposing the requirement into its smallest components. These are:

- Items in the order must be from the same restaurant.
- There must be between 1 and 4 (inclusive) items in the order.
- The card number must be valid.
- CVV must be valid.
- Card must be in date.

Some further breakdowns of testing and test-cases for this requirement available in the 'Order validation further detail' document

It will also aid in code visibility if the tests are broken down into smaller and more easily readable chunks.

This requirement is not necessarily integral to the core function of the system (drone flightpath); however, it is important for the real-world use of the system, and it is a reasonably sized task. This indicates that we should assign it a moderate number of resources and a medium-to-high priority.

R2: This is a system performance requirement so the pre-requisite for its testing is that the system has already been built. It is an important requirement for user satisfaction because runtime in excess of 60 seconds it can have knock-on effects for the restaurants making the orders and the users who expect them at certain times. Due to this it should be allocated a moderate priority but will require a low number of resources to test. This method of testing will improve the visibility with regards to late-stage quality evaluation.

R3: This is an integration level requirement which relies on the successful interaction of the REST server data retrieval and the No-Fly-Zone builder functionality. Integration of the system with the REST server is an important part of the overall system. This testing should have moderate priority but necessarily come after the unit testing of each individual component involved. A pre-requisite for testing is that we have a REST server up and running which we can pull the data from for our REST data retrieval functionality.

Scaffolding and instrumentation

R1: The actual tests will be unit tests for each sub-validation requirement which will each be given orders to test the normal, extreme and exceptional cases.

This requirement is fairly self-contained so does not need much scaffolding to test. We will need scaffolding to create orders with the desired attribute values for the test in question. This can be as simple as a template pasted into each test where each individual attribute is set. This approach may be inefficient; however, it improves visibility for the testers since you can see exactly what attribute has what value in the test rather than referring to another method/class/section of the file.

Instrumentation we will use for testing are the enum values of the validity attribute which is initialised to ValidButNotDelivered and assert statements checking the value of the validity field so assess whether the validation detected the intended order fault.

R2: This is a system performance requirement so will not require much scaffolding as the system should already be mostly built and it is a measurement of time.

We will need instrumentation in the form of a timer which we can start and stop at the beginning and end of the calculations and then get the time elapsed. This will likely be a small number of lines of code and won't take much time. However, we may need to test the instrumentation before applying it to **R2** which might require some mock 'system' which takes a known time to run which we can time with our instrumentation and compare to the expected result.

R3: This requirement will require scaffolding that can populate the pre-made REST server with files containing various no-fly-zones defined and in varying numbers.

We will also need a way for the results of the no-fly-zone builder to be output as a GeoJson file which we can then load into a service like geojson.io to visualise the outcomes of the tests.

Process and Risk

For this project we have chosen the DevOps lifecycle process.

R1: This requirement will be tested in the *create* stage of the DevOps process and the tests will be used in the verification stage as well. Once deployed, statistical data on the real-world performance of the validation can be gathered and used to identify weaknesses that our tests didn't, which we can then plan for, test for and fix and begin the cycle again.

The orders must be validated thoroughly for real world use which causes risk since our testing relies on human identification of test cases to cover. If there are omissions in edge case testing for example, then the system may break when operating for real orders. Minimising this occurrence will require the allocation of more resources (i.e., people to quality control, time, creating better synthetic data generators).

R2: Within the DevOps lifecycle this requirement testing will be mostly used in the verifying and monitoring stages where the performance data can be used to assess the quality before deployment and to inform on the performance during real-world use.

Risks associated with this requirement testing is that the performance data may not be representative of real world performance so on day one of deployment there may be very large runtimes which make the system unavailable or cause orders to be significantly delayed. The risk of this should not be *too* high if we give the system performance tests datasets of the theoretical maximum number of orders the system could handle. However, there could be some other contributing factor not identified in testing so there is still enough risk to warrant allocating some resources to minimising it.

R3: The initial requirement testing should be scheduled for the mid-to-second half of *create* stage testing and be given a reasonable length of time to allow for fixing any errors that may occur.

One risk is that the integration testing will come too late due to having to wait for the component development and unit testing. If either of these run over time budget it can limit the ability of the testers to resolve integration issues that arise. The testing also relies on the REST server to be operational, so another risk is that the server has unforeseen downtime or other errors.

***<u>Evaluation</u>*:**

For **R1** the test plan was followed virtually to the letter even in a process and environment which was evolving as the development went on. This speaks to the quality of the plan being high as it withstood the unpredictability of real world development and proved to be the course of best fit. The plan identifies the main risks the test approaches and indicates some areas to which allocating resources may help, however the named risks are not exhaustive.

For **R1**, breaking the requirement into its smallest testable sub-units aided in creating a high volume of tests which were able to give good coverage for the code and input cases. However, the final testing was limited in that it tested with single orders. The testing could be improved and be more representative if it included tests with larger datasets. The planned instrumentation was used and proved to be adequate for evaluating the requirements.

In reality, **R2** was not able to be tested as the system was not completed due to coding difficulties and time restraints in ILP so there are instead described tests in the repository. The testing I believe would have been implemented very similarly to the test plan as is described in the repository. The planned instrumentation is core to this requirement testing; however, the testing may have been aided by further instrumentation to collect statistical data on averages or extrapolations of times recorded over multiple tests and test cases.

For **R3**, the test plan was accurate in that it identified a risk that came to fruition during testing which was unforeseen issues with the REST server. I was unable to set up a way to create REST server files to use in testing and instead had to rely on one pre-established but with limited data sets.

The results to GeoJson instrumentation was implemented which aided in assessing with more confidence the verification of the requirement, especially with the lack of diverse REST data.

Overall, the test plan was representative of the actual testing and a useful tool for guiding the process of verifying the requirements, though it does have omissions.