Identifying testing approach for chosen attributes:

Functionality requirements implementation strategy:

In order to verify that whole system works as expected, we need to ensure that each unit of the system has been tested, so unit testing will be utilized for given requirement.pdf doc.

Once each unit has been tested, we need to ensure that integration between modules are handled as required we integration testing will be done for Rest service interaction with other modules of the system such as, does flightpath class fetches data properly and from those fetched data does it process it and pass the tests? These will be done after unit test.

System-level testing will also be done to confirm that system as a whole works correctly. From fetching data from Rest Service all the way down to file creation. Also, for safety and privacy compliance we ensure the system complies with safety and privacy requirements, possibly through scenario-based testing.

I influenced my testing methodology based on the analysis and testing principles presented in Chapter 3. Static type checking and asserting properties in the code are unnecessary but will help to build confidence. The requirements will be limited to reduce complex problems into simpler ones. To make testing easier, the more complex test will be broken down into smaller parts. In order to make sure that every part of the system is handled, I have introduced code coverage, specifically function and condition coverage will be used. Finally, I will be introducing print statements all over the system itself and tests itself to help with testing. As well as that scaffolding will also be installed.

Non-Functional Requirements implementation strategy:

We have covered wide range of system level requirements in non-functional part. After the system is finished, these will be verified by non-functional system level testing and black box testing. To assess the system's performance, timing testing will be utilized. Stress testing will be done to test if system can handle enormous load of data. As well as that, security testing will be employed to test whether Rest API has required certificates and https connection has been ensured.

My main approach to test the system will be:

Out of black-box, white-box, grey-box, DevOps continuous testing. I will be choosing grey-box testing where grey-box testing:

This is a hybrid of the white-box and black-box methodologies. This method basically offers the advantages of getting to test underway early and enabling time at the end to fine-tune/design test cases with regard to implementation specifics. With this method, we mostly rely on grey-box testing, in which some implementation details are known.

Where black-box testing: Rather than focusing on the code itself, this method is focused with checking the specification (functional requirements). Without understanding the internals, such as how the final user might perceive the result, it concentrates on the output of the input. This approach is integrating the testing process in the development stage also called proactive approach.

Where white-box testing: When a developer is familiar with the structure and content of the code, they can do structural testing. Usually, it enhances the above-mentioned black box testing. We are worried about if the entire code is executed, utilized, and produces particular tests upon code implementation. This approach also performing testing analysis once the system is fully implemented also called reactive approach.

Appropriateness of Chosen Testing Approach

We know that unit tests are to separate tiny parts of the code and it checks if the code works as required. Unit tests makes integration easier due to its nature, it can detect bugs early in the implementation of the application and this makes it simplifies refactoring existing code. As well as that unit tests makes debugging operation and it makes sure that the code implemented had high cohesion because it supports modular programming. But there are some down sides of unit testing, we know that unit tests are costly and it consumes time like nothing, also it doesn't guarantee us to detect "all" the bugs.

On the other hand, integration tests are checking if the modules are interfacing correctly. Integration tests helps detecting integration problems between connected components of the system and it makes sure that every component of the system works as expected before going into next stage "system level" testing. However, while integration tests check the interactions between components, they may not fully capture the end-to-end user experience or system performance under real-world conditions.

System testing is good for testing the application as a whole, from fetching orders to file creation. It makes sure that the system meets the all the requirements stated in specification However, this testing might not be good to uncover all performance-related issues and security vulnerabilities. Also, system testing can be time-consuming and really costly.

Security testing is vital for an this system to handle sensitive user data. It focuses on input validation, secure connections (HTTPS), and SSL certificate validations. However, security testing as described might not cover all types of security threats. For instance, it might not fully address issues like cross-site scripting or advanced persistent threats. Tests do not include SQL injection resistance tests because we know the implementation will not use a database.

Because this system must operate in real-time, timing tests are crucial to both user pleasure and ensuring that the system replies in a reasonable amount of time. These tests, however, might not take into account all the factors—like hardware constraints or network latency—that effect timing in a real-world setting.

For our main approach of testing:

Time, flexibility, and affordability are three important areas where this strategy advantages above the others. Proactive testing requires a large initial time investment, with additional time later required for test suite improvement, which may have an impact on the quality of the finished product. On the other hand, last-minute reactive testing frequently results in subpar test suites with errors that are only found later. In terms of resources, the proactive method increases initial resource commitment and overhead by requiring significant early attention. Reactive approaches, which concentrate attention after development, run the risk of late-stage bug detection and possible delays in product release because of labor-intensive remedies.

It is important to mention that while DevOps continuous testing strategy is theoretically the best approach, it is not practical for this project and will not be taken into consideration because it doesn't scale up enough. It would have been a good choice if the system was bigger.