In Your Orbit

Software Testing Plan

Version 1.0

Cloud-Based Planetary Ephemerides

United States Geological Survey

Mentor: Scott Larocca

Austin Carlile
Noah Schwartz
Minuka Trikawalagoda
Nicholas Gonzalez

The purpose of this design document is to explicitly detail the testing plan of our software solution

3/26/2025

Table of Contents

1. Introduction 3

2. Unit Testing 5

3. Integration Testing 9

4. Usability/End-User Testing 14

5. Conclusion 16

1. Introduction

Project Overview

Planetary science, driven by NASA missions, relies heavily on processing planetary images using sensor models and Image Support Data (ISD) from NASA's complex SPICE system. However, accessing ISD is often difficult and time-consuming.

To simplify this, our capstone team is building a cloud-based web service by leveraging USGS tools and AWS that streamlines ISD retrieval without requiring massive SPICE downloads. The system is designed to be scalable, user-friendly, and compliant with NASA protocols.

By improving access to ISD, our project will enhance image processing workflows for NASA and USGS, and lower barriers for new researchers in the field.

Purpose of Testing

Software testing is the process of evaluating a software application to ensure it functions as intended and meets user requirements. It involves checking for bugs, verifying that features work correctly, and validating overall system performance. Effective testing helps catch issues early, improves reliability, and ensures a better user experience.

In our project, software testing is especially important due to the complexity of retrieving and processing large scientific datasets. By thoroughly testing our web service, we can ensure that ISD retrieval is accurate, efficient, and reliable which is critical for supporting planetary science research. Testing also helps us maintain performance, security, and compliance with NASA and USGS standards as the system scales.

Overview of Testing Approach

We will be conducting unit testing, integration testing, and usability testing to ensure the functionality, reliability, and user-friendliness of our cloud-based ISD retrieval system. Unit testing will focus on core components such as ISD generation, caching logic, and API endpoints, ensuring that individual functions behave as expected. Integration testing will verify that our system components including but not limited to, the FastAPI web service, caching server, and the ALE-based ISD generation engine, which work together seamlessly. Usability testing will assess the user interface and overall experience for scientists accessing ISDs, focusing on ease of use and responsiveness.

The scope of our testing will be moderate to high, with automated unit and integration tests covering all critical paths and failure cases. Usability testing will be lightweight but targeted, involving a small group of domain-relevant users.

We will use various testing tools for unit and integration tests including but not limited to pytest, unittest.mock, FastAPI's built-in test client, and mocking tools to isolate components. For usability, we will use observation-based testing in sandboxed environments via Docker, supplemented by informal feedback and walkthroughs.

Our rationale when it comes to the kinds of testing we are doing and what tools to use stem from the specific work we are doing for the client. Our solution is a Python program that utilizes FastAPI and is hosted on AWS services so the testing reflects that. We are using FastAPI’s testing client and Docker as that is the best way to isolate the environment in which we are operating our software stack. Additionally, our usability and end-user testing looks a little different as this solution is for a very specific client and as such user testing does not work with anybody with a computer, only those scientists and computer engineers who understand the complex ALE and SPICE database.

The following sections will go into much further detail about the different methods of testing we utilized for our project including unit testing, integration testing, and some limited end-user testing.

2. Unit Testing

Overview

Unit testing is the practice of testing individual units of code—functions, methods, or classes—in isolation to verify that they perform as expected. In our cloud-deployed ISD retrieval system, unit testing plays a critical role in ensuring correctness before deploying services that rely on external systems such as AWS DynamoDB, the SPICE toolkit, or file I/O operations.

Goals of Unit Testing

The primary goals of unit testing in this project are to:

- Validate core ISD processing and hash generation logic.
- Ensure API endpoints behave as expected before integration.
- Prevent regressions as cloud deployment scales.
- Confirm robustness when handling various forms of label input (e.g., brotli-compressed, plain text).

Testing Tools and Metrics

To streamline our unit testing activities, we will use the following tools:

- **PyTest**: Python’s de facto standard for writing clear and scalable unit tests.
- **Coverage.py**: To measure how much of our codebase is exercised by tests.
- **FastAPI TestClient**: To isolate and test individual API endpoints.
- **unittest.mock**: To simulate external systems such as databases or caches.

We aim to achieve:

- **>85% code coverage** for isdAPI.py
- Tests must pass in local and cloud environments
- Unit tests must complete in under 5 seconds on average

Units to be Tested

The following are the core units identified for testing within the system. Each unit has been selected based on its criticality to the ISD processing workflow, its interaction with external services, or its role in ensuring data correctness and consistency during cloud deployment.

Unit	**Description**	**Why Test**
create_mini_label	Parses an ISIS label file and extracts key fields into a compact dictionary structure	Ensures correct metadata is extracted for ISD identification and hashing
create_hash	Creates a SHA-256 hash of the mni-label string to uniquely	Verifies that identical labels generate consistent hashes,

	identify the ISD	and different labels yield different hashes
POST /getIsd	Accepts compressed label files, decompresses, runs ISD generation, stores/retrieves from DynamoDB	Central endpoint for users; validates full processing pipeline from input to storage to output
POST /create	Accepts a hash and ISD dictionary and stores it in DynamoDB	Confirms manual ISD creation behaves consistently with automatic pipeline storage
GET / read/{id}	Retrieves a stored ISD entry from DynamoDB by its hash	Ensures correct retrieval of existing records and graceful failure on invalid or missing keys
PUT /update/{id}	Updates a stored ISD entry with new values (through current implementation does not match Item)	Validates update operations function correctly and syntax is consistent
DELETE /delete/{id}	Removes an ISD entry from DynamoDB using its ID	Ensures that entries can be deleted reliably and no data is left behind

Rationale for Selected Units

Unit testing will be focused on:

- **Data transformation and hashing logic**, since incorrect hashes can lead to cache misses.
- **Endpoint handlers**, to confirm that they accept and process input correctly before integration.

External tools like spiceinit and isd_generate are **not unit tested directly** due to their complexity and reliance on external system state (these are better covered in integration tests). Instead, we mock their output where needed.

Test Design and Plan

Function / Endpoint	**Valid Cases**	**Boundary Tests**	**Error Cases**
create_mini_label	Full ISIS3 label with all required fields	Minimal label, extra unused fields	Missing groups, malformed PVL, invalid file path
create_hash	Standard mini-label strings	Very short or long strings	Empty string, non-string input
decimalToFloat	Decimal values in JSON	Nested decimals	Unsupported types (e.g., datetime)

POST /getIsd	Proper Brotli-compressed label	Large label files	Corrupted compression, missing fields, subprocess or DB failure
POST /create	Valid hash and ISD dictionary	Very large/small payloads	Malformed JSON, duplicate keys
GET /read/{id}	Known existing hash ID	Short/long valid hashes	Nonexistent ID, invalid ID format
PUT /update/{id}	Valid update body with ID	Partial updates	Missing fields, schema mismatch, ID not found
DELETE /delete/{id}	Deletion of existing ID	Very short or long IDs	ID not found, invalid format

3. Integration Testing

Overview

- What integration testing is

Integration testing is a critical phase in software verification that ensures the various modules of a system communicate and operate together correctly. While unit tests validate individual functions or methods in isolation, integration tests focus on validating the interactions and data exchanges across module boundaries. This is particularly important in distributed or microservices-based systems such as ours, where different components handle web requests, perform caching, manage databases, or generate ISDs using external tools.

- Goals of integration testing

The primary goal of integration testing in our project is to verify the correctness of communication between the following subsystems:

- ❖ The Web Service and Caching Server
- ❖ The Caching Server and ISD Generation Engine
- ❖ The Caching Server and SPICE Database
- ❖ The Web UI and FastAPI Web Service
- ❖ The AWS infrastructure hosting the components

This phase ensures that requests from users are processed end-to-end correctly — from UI interaction through API handling, ISD generation, caching, and return of data. It also confirms that all modules respect interface contracts, such as correct data formats and error handling protocols.

- Overall strategy: top-down, bottom-up, or sandwich?

Given the structure and flow of our system, we used a bottom-up integration strategy. This allowed us to test low-level integration points (e.g., ISD generator ↔ cache) before higher-level ones (e.g., FastAPI ↔ client). This was particularly useful for catching issues in core data processing logic before layering network or client interaction logic.

Tools and Frameworks Used

- ❖ **Postman**: For API interaction testing
- ❖ **Docker Compose**: For setting up test environments involving multiple containers
- ❖ **Pytest**: For scripting automated integration tests
- ❖ **AWS CloudWatch Logs**: For verifying system events and function execution
- ❖ **Mock Services / Fakes**: For simulating SPICE updates and network failures

Integration Points

List and describe testing for each major integration point:

- Database interactions

  What is being tested

  The interaction between the **Web Service**, **Caching Server**, and **Amazon DynamoDB/ElastiCache** to store, retrieve, and update ISDs.

  Tools/Test Harnesses Used

  - ❖ Postman for issuing GET requests
  - ❖ Pytest with boto3-mock to simulate DynamoDB and ElastiCache responses
  - ❖ Docker for isolated database testing

  Success Criteria

  - ❖ Data is correctly written to and read from the database
  - ❖ ISDs are returned accurately based on hash-based queries
  - ❖ Cache invalidation occurs when SPICE updates are simulated

- API calls

  What is being tested

  The connection between **FastAPI** and the back-end logic (ISD generator and cache), including response correctness and latency.

  Tools/Test Harnesses Used

  - ❖ Postman and Pytest HTTP clients
  - ❖ Dockerized test environments
  - ❖ Log checks in AWS CloudWatch

  Success Criteria

  - ❖ Correct HTTP response codes and payloads are returned
  - ❖ Latency remains below the defined threshold for common request types
  - ❖ Proper fallback behavior when cache misses occur (triggers ISD generation)

- UI ↔ Logic layers

  What is being tested

  Communication between the **user interface** and **FastAPI endpoints**, focusing on the validity of requests and responses.

  Tools/Test Harnesses Used

  - ❖ Manual interaction with UI prototypes

    - ❖ Browser DevTools + Postman
    - ❖ JSON schema validation

  Success Criteria

    - ❖ All UI input fields map correctly to API parameters
    - ❖ API responses are rendered correctly in the UI
    - ❖ Errors (e.g., invalid label file) are properly surfaced to the user

- **Any third-party services**

  What is being tested

  Integration of our ISD generation logic with **NASA's ALE tool** and the **SPICE database**, including correct parsing and ISD formatting.

  Tools/Test Harnesses Used

    - ❖ Local test scripts that simulate .lbl file input and call isd_generate()
    - ❖ Simulated SPICE kernel updates using version-controlled files
    - ❖ Docker to isolate ALE interactions

  Success Criteria

    - ❖ Valid ISD JSON files are generated from input labels
    - ❖ ISDs are re-generated when SPICE data updates
    - ❖ Hashes reflect unique mini-labels for different input parameters

4. Usability and End-user Testing

Overview

Usability Testing, also known as End-User Testing, is the process of testing a product or service from the perspective of an end-user. This type of testing is exclusively concerned with the functionalities, as well as the User Interface, that the end-user will directly interact with. The goal of Usability Testing is to make sure the product or service is easy to understand and use for the user.

Our Web Service and Caching solution will be primarily used by Planetary researchers, scientists and students. While working within the Planetary Sciences field requires a high level of technological knowledge and ability, our users will have little-to-no experience with the specific systems required to generate ISDs. For this reason, we need our User Interface and User Experience to be as straightforward and easy-to-use as possible, even though a small level of technological ability is fundamentally required for operating this service.

Our solution will only present two interface functionalities to the user, a getIsd() function, requiring them to specify a label file that will be sent to the Web Service; and a returned_isds folder, that will contain the generated ISDs that have been returned from the Web Service. This implementation has the fewest amount of moving parts possible and only requires the user to know the name of the label file that they are sending to the Web Service, as well as the file location of the ISDs when they are returned. In order for our system to be easy to understand and use, we will need to test and refine our user experience, to ensure that the label specification and ISD file locations are straight forward with clear explanations of use, requiring as little background information as possible.

Testing Plan and Rationale

The Usability Tests that we will be leveraging to test our solution will be Expert Reviews. Focus Group Testing is not feasible for our project because we do not have access to large groups of Planetary Scientists, and cannot secure a practical test group within the time constraints of our project. Similarly, User Observation Testing is also not feasible because it would require finding a test user and sending them the solution, then teaching them how to set up and use it, as well as set up the ALE and ISIS environments to run the solution locally. The coordination required to accurately run such tests in the given time frame is unrealistic. Expert Review Tests are the most feasible tests for our project, since we have access to our Client, the USGS Astrogeology team, that contracted our solution and knows the exact User Experience that is needed. For these reasons we will be using Expert Review Tests to ensure that the User Experience is easy to understand and use, and functions as required.

We will be conducting Expert Reviews using stakeholders that will have the same foundational knowledge that our end-users will. The specific stakeholders that we will be conducting our reviews with will be members of the USGS Astrogeology team that contracted our product solution. We will have two participants, Kelvin Rodriguez, the Astrogeology team’s lead; as well as Amy Stamile, a software developer on the Astrogeology team. We will be conducting our Usability Test from April 15th until April 20th, allowing us to gain a sizable amount of Expert Review data and implement it, refinding our Usability and User Experience before delivering the final product to our Client.

Data Collection and Analysis

We will be collecting data from our Usability Tests using in-person interviews with the Expert Reviewers. Our Expert Reviewers will interact with our Web Service and provide us with feedback on its User Experience and User Interface Functionality in real time. On top of this, we will be asking the follow-up questions:

1. How much did you need to learn before you could use the Web Service properly?
2. How would you describe the difficulty of requesting an ISD from the Web Service?
3. How would you describe the difficulty of using the Web Service? How would you describe the difficulty of finding the ISDs?
4. What aspects of using the Web Service did you find difficult to understand or unintuitive?
5. What aspects of using the Web Service did you find easy to understand or intuitive?

We will analyze the data by reviewing which parts of the User Experience that the Expert Reviews found easy to understand and use and which part they found difficult. From there, we can identify the aspects of our solution's Usability that need to be improved and which aspects were satisfactory. Then we can compare the implementations between the satisfactory and subpar aspects, giving us a guideline to develop and improve the User Experience for our end users.

5. Conclusion

Software testing is an integral part of ensuring that a product functions correctly and meets user requirements and needs. Due to the complex nature of the problem that we are solving, retrieving and processing extremely large amounts of data, Software Testing is vitally important to ensuring that we deliver a successful product. We will be utilizing Unit, Integration, and Usability testing to ensure the functionality, reliability, and user-friendliness of the solution. Our Unit Tests will focus on core components, making sure that individual functions behave as expected, while our Integration Tests will verify that our various system components fit and work together seamlessly. Finally, our Usability Tests will leverage Expert Reviews to assess the End-User Experience, ensuring a solution that is both easy to understand and use. The testing plan laid out in this document gives us a concrete way to not only test and make sure that our solution functions correctly but to also ensure that it meets our end-user’s needs and requirements, allowing us to deliver a robust and polished solution that is easy to understand and use.