Software Verification and Validation Plan

D-XXX

EOX Team

Version 0.1, 11/12/2023

DESIDE - Software Verification and Validation Plan

COMMENTS and ISSUES If you would like to raise comments or issues on this document, send an email to <office@eox.at>.	**PDF** This document is available in PDF format here.
EUROPEAN SPACE AGENCY CONTRACT REPORT The work described in this report was done under ESA contract. Responsibility for the contents resides in the author or organization that prepared it.	**EOX IT Services GmbH** Thurngasse 8/4, 1090 Vienna, Austria. eox.at

AMENDMENT HISTORY

This document shall be amended by releasing a new edition of the document in its entirety. The Amendment Record Sheet below records the history and issue status of this document.

Table 1. Amendment Record Sheet

ISSUE	DATE	REASON
0.1	11/12/2023	Initial in-progress draft
1.0		First released version

Chapter 1. Introduction

1.1. Purpose and Scope

This document represents the Software Verification and Validation Plan (SVVP) for the DESIDE project. This document describes generic regression/unit tests that are run on the software when new commits are performed to ensure the software is still functioning as expected.

1.2. Structure of the Document

Section 2 - Overview

This section provides an overview of the DESIDE

Section 3 - Verification

This section provides the software verification and validation plans, activities, resources, acceptance criteria, schedule and change control.

1.3. Reference Documents

The following is a list of Applicable and Reference Documents with a direct bearing on the content of this document.

Reference	Document Details	Version
[SOW]	Statement of Work Destination Earth DESP Use Cases selection - Round 1 Reference: CS301353.Docref.0002	1.0
[Proposal]	Proposal No. 8482: DestinE Sea Ice Decision Enhancement (DESIDE)	1.1 06/06/2023

1.4. Terminology

The following terms have been used in this document.

Term	Meaning
Admin	User with administrative capabilities on a platform.
Code	The codification of an algorithm performed with a given programming language - compiled to Software or directly executed (interpreted) within the platform.
Discovery	User finds products/services of interest to them based upon search criteria.
Interactive Web Application	An Interactive Application for analysis provided as a rich user interface through the user's web browser.

Term	Meaning
Key-Value Pair	A key-value pair (KVP) is an abstract data type that includes a group of key identifiers and a set of associated values. Key-value pairs are frequently used in lookup tables, hash tables and configuration files.
Object Store	A computer data storage architecture that manages data as objects. Each object typically includes the data itself, a variable amount of metadata, and a globally unique identifier.
Products	EO data (commercial and non-commercial) and Value-added products.
Software	The compilation of code into a binary program to be executed within the platform on-line computing environment.
User	An individual using the services.
Visualization	To obtain a visual representation of any data/products held within the platform - presented to the user within their web browser session.
Web Coverage Service (WCS)	OGC standard that provides an open specification for sharing raster datasets on the web.
Web Feature Service (WFS)	OGC standard that makes geographic feature data (vector geospatial datasets) available on the web.
Web Map Service (WMS)	OGC standard that provides a simple HTTP interface for requesting geo-registered map images from one or more distributed geospatial databases.
Web Map Tile Service (WMTS)	OGC standard that provides a simple HTTP interface for requesting map tiles of spatially referenced data using the images with predefined content, extent, and resolution.
Web Processing Services (WPS)	OGC standard that defines how a client can request the execution of a process, and how the output from the process is handled.

1.5. Glossary

The following acronyms and abbreviations have been used in this document.

Term	Definition
ADD	Architecture Design Document
AOI	Area of Interest
API	Application Programming Interface
COG	Cloud optimized GeoTiff
EO	Earth Observation
EOX	EOX IT Services GmbH
ESA	European Space Agency
FUSE	Filesystem in Userspace

Term	Definition
ICD	Interface Control Document
JSON	JavaScript Object Notation
KVP	Key-value Pair
M2M	Machine-to-machine
OGC	Open Geospatial Consortium
REST	Representational State Transfer
SDD	Software Design Document
SFTP	Secure File Transfer Protocol
SRF	Software Reuse File
SRN	Software Release Note
SRP	Software Release Plan
SRS	Software Requirements Specification
SSH	Secure Shell
STAC	Spatio-Temporal Asset Catalog
SUM	Software User Manual
SVVP	Software Verification and Validation Plan
SVVR	Software Verification and Validation Report
TOI	Time of Interest
UMA	User-Managed Access
US	User Story
WCS	Web Coverage Service
WFS	Web Feature Service
WMS	Web Map Service
WMTS	Web Map Tile Service
WPS	Web Processing Service
WPS-T	Transactional Web Processing Service

Chapter 2. Overview

This section provides an overview of the DESIDE DESIDE. It highlights …

Chapter 3. Verification

The verification approach for the DESIDE system consists of a combination of unit testing, integration testing, and system testing.

Unit Testing: Each component of the data pipeline undergoes thorough unit testing. Unit tests are designed to verify the individual functionality of each component in isolation. The unit tests ensure that the components perform as expected and adhere to the defined requirements.

Integration Testing: Integration testing is conducted to verify the interactions and compatibility between the components of the data pipeline. Integration tests are executed to ensure proper data flow and integration points between the components. These tests focus on verifying the overall functionality and communication of the integrated components.

Server Testing: The server responsible for data sharing is subjected to a comprehensive set of tests. These tests cover various aspects, including data input/output verification, data storage and retrieval, error handling, and performance under different load conditions. The server tests are designed to ensure the reliability, stability, and efficiency of the data sharing functionality.

System Testing: The main repository, which contains the deployment and bundling system, is verified through system testing. System tests are designed to evaluate the end-to-end functionality and behavior of the software system as a whole. These tests cover various scenarios and use cases to ensure that the system operates as intended and meets the specified requirements.

To facilitate the testing process, the `pytest` framework has been selected as the primary testing tool. `pytest` offers ease of use, is widely adopted within the industry, and provides comprehensive documentation. Its rich set of features enables efficient test development, execution, and result analysis.

The rationale behind this approach is to ensure that each component of the software system is thoroughly verified in isolation, as well as in conjunction with other components to verify their integration. By adopting a combination of unit testing, integration testing, and system testing, we aim to identify and address any issues early in the development cycle, ensuring the delivery of a high-quality and reliable software system.

3.1. Verification activities

This section outlines the verification activities for key DESIDE components.

3.1.1. TBD

3.2. Verification criteria and acceptance

Types of verifications performed:

- Tests
- Demonstration

3.2.1. Test

Test Execution: The primary acceptance criteria for verification is that all tests, including unit tests, integration tests, and system tests, pass successfully without any critical failures or errors.

Test Results: The verification process will consider the test results generated from the execution of the test suite. The results should indicate a high percentage of passed tests, demonstrating that the software system meets the expected functionality and behavior.

Error Handling: The software system should exhibit appropriate error handling mechanisms. Verification will verify that error messages are displayed accurately, and the system recovers gracefully from errors without causing any data loss or instability.

3.2.2. Demonstration

In addition to testing, demonstration will be conducted as part of the verification process. The demonstration aims to showcase the functionality, features, and capabilities of the software system in a real or simulated environment.

By including a demonstration as part of the verification process, we aim to provide stakeholders with a tangible and visual representation of the software system's capabilities. The demonstration serves as an effective means to validate the software against the specified requirements and ensure that it meets the expectations of the end-users and stakeholders.

3.3. Verification resources

This section outlines the resources which are utilized for conducting of the verification activities.

The main resources are:

1. GitLab Runner
2. GitHub Actions

These resources offer the necessary infrastructure and automation capabilities to execute the verification activities effectively. The use of a self-hosted GitLab Runner and GitHub Actions ensures reliable and controlled environments for conducting the tests and analyzing the results.

3.3.1. GitLab Runner

The ETL components and system testing utilize a self-hosted GitLab Runner for test execution. The self-hosted GitLab Runner provides a dedicated environment for running tests and ensures consistent and controlled testing conditions. The configuration and management of the GitLab Runner are handled internally by the project team.

3.3.2. GitHub Actions

Some testing activities make use of GitHub Actions for test execution. GitHub Actions provide an automated workflow and testing environment for running tests on the server. The GitHub Actions workflow is configured and maintained within the project's GitHub repository.

3.4. Verification change control

Change control procedures are implemented to manage any changes to the software system during the verification process. This ensures that changes are properly evaluated, documented, and approved to maintain the integrity of the verification effort. The following steps outline the change control process:

Change Identification:

- Any proposed change to the software system is identified and documented.
- Changes can include modifications to requirements, design, code, or any other aspect of the system that may impact verification.

Change Impact Assessment:

- The impact of the proposed change on the verification effort is assessed.
- The verification team evaluates the potential effects of the change on test plans, test cases, test data, verification schedule, and other relevant aspects.
- The assessment considers the potential risks and benefits of implementing the change.

Change Approval:

- The proposed change is reviewed and approved by the designated change control authority.
- The approval decision is based on the impact assessment, project priorities, and alignment with the overall project goals.
- The change control authority may consist of project managers, stakeholders, or a designated change control board.

Change Implementation:

- Once the change is approved, it is implemented according to the established procedures.
- The necessary modifications are made to the affected artifacts, such as test plans, test cases, or verification documentation.
- The verification team ensures that the necessary updates are communicated to all relevant stakeholders.

verification Impact Review:

- After implementing the change, the verification team reviews the impact of the change on the verification activities.
- Test plans, test cases, or any other affected verification artifacts are updated to reflect the changes.
- The verification team verifies that the implemented change does not adversely affect the overall verification process or compromise the quality of the software system.

Effective change control procedures help maintain the integrity and reliability of the verification effort by ensuring that any changes to the software system are properly evaluated, documented,

and incorporated into the verification process. These procedures help minimize risks associated with changes and ensure that the verification remains aligned with the project goals.

3.5. Verification schedule

The software verification activities will be conducted in a flexible manner that accommodates the dynamic nature of the development process. While there is no fixed schedule for the verification activities, the following approach is adopted:

1. Iterative verification: The verification process is integrated into the development iterations or cycles. As each component or feature reaches a stable state, verification activities are performed to ensure its functionality, performance, and compliance.
2. Continuous Integration: The verification activities are integrated into the continuous integration and delivery (CI/CD) pipeline. Automated tests are triggered upon code changes, ensuring that the software system is continuously verified as new features or updates are introduced.
3. Trigger-Based verification: verification activities may be triggered by specific events, such as significant changes in the software system, updates to dependencies, or critical bug fixes. These triggers initiate a focused verification effort to ensure the stability and reliability of the affected areas.
4. verification on Demand: verification activities may be requested by stakeholders, such as project managers, clients, or regulatory bodies. These requests are addressed promptly, and verification efforts are planned and executed accordingly.

The absence of a fixed schedule allows for a flexible and adaptable approach to software verification, enabling the verification activities to align with the evolving nature of the software development process. The verification plan is continuously updated to incorporate any changes or adjustments to the verification timeline.

3.6. Code quality

Code quality is a crucial factor in the software verification process, ensuring that the codebase is well-structured, maintainable, and adheres to industry best practices. To achieve and maintain high code quality standards, the following measures are implemented:

1. Code Review: All code changes undergo rigorous code reviews by experienced developers or designated code reviewers. Code reviews help identify and address any potential issues related to code style, logic errors, performance optimizations, and adherence to coding guidelines and standards.
2. Automated Testing with Flake8 and Mypy: Automated testing tools such as Flake8 and Mypy are employed to analyze the codebase for potential issues. Flake8 is a linting tool that checks for coding style violations, potential errors, and adherence to coding conventions. Mypy is a static type-checking tool that helps identify type-related errors and inconsistencies. These tools are integrated into the CI/CD pipeline to ensure that code changes adhere to coding standards and maintain consistent code quality.
3. Code Formatting with Black: During the development process, the codebase adheres to a consistent code style using the Black code formatter. Black automatically formats the code to

ensure uniformity in code style and readability. By utilizing Black, the codebase has consistent formatting, minimizing style-related discrepancies and improving code maintainability.

4. Test Coverage: Adequate test coverage is maintained to ensure comprehensive testing of critical parts of the codebase. This includes unit tests, integration tests, and system tests. Test coverage metrics are monitored to identify areas with insufficient coverage, allowing for targeted improvements to increase the overall code quality and reliability.
5. Continuous Integration and Continuous Deployment (CI/CD): A CI/CD pipeline is established to enforce automated code quality checks. This includes running Flake8 and Mypy as part of the pipeline to identify and report any coding style violations, potential errors, or type inconsistencies. Only code changes that pass the defined code quality criteria are deployed to the target environment.
6. Refactoring and Code Maintenance: Periodic refactoring and code maintenance activities are conducted to improve code quality, enhance readability, and address any technical debt. Refactoring efforts are planned and executed to minimize disruptions and ensure the ongoing stability of the software system.

By incorporating the use of Flake8, Mypy, and Black within the automated testing and development process, we aim to enforce coding standards, identify potential errors and inconsistencies, ensure consistent code style, and maintain high code quality throughout the software development lifecycle.

3.7. Performance verification

In addition to functional testing, performance testing is an integral part of the software verification process. Performance testing aims to assess the software system's response time, scalability, and stability under various load conditions. Locust, an open-source performance testing tool, is utilized to conduct performance testing. The following points outline the approach for performance testing using Locust:

1. Test Scenarios and Workloads: Test scenarios are defined to simulate realistic user behavior and workload patterns. These scenarios mimic different types of user interactions, such as browsing, searching, and data processing, to represent real-world usage patterns. Workloads are designed to reflect expected user loads, including the number of concurrent users, requests per second, and data volumes.
2. Load Generation with Locust: Locust is used to generate virtual user load and simulate concurrent user interactions. With Locust, we can create test scripts using Python to define user behaviors, request patterns, and response verifications. The tool allows us to distribute the load across multiple machines to simulate high traffic conditions.
3. Metrics and Monitoring: During performance testing, various metrics are collected, such as response times, throughput, error rates, and resource utilization. Additionally, server-side monitoring tools may be utilized to capture system-level metrics, including CPU usage, memory consumption, and network activity. These metrics provide insights into the performance characteristics of the software system under different load conditions.
4. Analysis and Tuning: The performance test results are analyzed to identify any bottlenecks, performance degradation, or scalability issues. Based on the analysis, necessary optimizations

and tuning activities are performed to improve the software system's performance. This may involve adjusting configuration parameters, optimizing database queries, or enhancing system architecture.

5. Verification Thresholds: verification thresholds for performance testing are established based on defined performance objectives and requirements. These thresholds define the acceptable performance levels for key metrics, such as response time, throughput, or error rates. The performance test results are compared against these thresholds to determine if the software system meets the performance expectations.

Proposed scenarios:

Cache tests:

- test wms using (1 workers, 100 users, 100 hatch rate)
- test wmts using (1 workers, 100 users, 100 hatch rate)

Renderer tests:

- WMS 10 users + 10 hatch rate
- WMS 50 users + 50 hatch rate
- WMS 100 users + 100 hatch rate
- WCS 10 users + 10 hatch rate
- WCS 50 users + 50 hatch rate
- WCS 100 users + 100 hatch rate

By leveraging Locust for performance testing, we aim to evaluate the software system's performance under realistic load conditions and identify any performance-related issues. The insights gained from performance testing guide optimization efforts and ensure that the software system performs optimally and meets the defined performance criteria.

<< End of Document >>