System Verification and Validation Plan for Plutos

Team #10, Plutos
Payton Chan
Eric Chen
Fondson Lu
Jason Tan
Angela Wang

November 1, 2024

Revision History

Date	Version	Notes
11/01/2024	1.0	Add: 3.1, 3.2, 3.3, 3.6
Date 2	1.1	Notes

[The intention of the VnV plan is to increase confidence in the software. However, this does not mean listing every verification and validation technique that has ever been devised. The VnV plan should also be a **feasible** plan. Execution of the plan should be possible with the time and team available. If the full plan cannot be completed during the time available, it can either be modified to "fake it", or a better solution is to add a section describing what work has been completed and what work is still planned for the future. —SS]

[The VnV plan is typically started after the requirements stage, but before the design stage. This means that the sections related to unit testing cannot initially be completed. The sections will be filled in after the design stage is complete. the final version of the VnV plan should have all sections filled in. —SS]

Contents

List of Tables

[Remove this section if it isn't needed —SS]

List of Figures

[Remove this section if it isn't needed —SS]

1 Symbols, Abbreviations, and Acronyms

symbol	description
T	Test

[symbols, abbreviations, or acronyms — you can simply reference the SRS (Author, 2019) tables, if appropriate —SS]

[Remove this section if it isn't needed —SS]

This document ... [provide an introductory blurb and roadmap of the Verification and Validation plan —SS]

2 General Information

2.1 Summary

[Say what software is being tested. Give its name and a brief overview of its general functions. —SS]

2.2 Objectives

[State what is intended to be accomplished. The objective will be around the qualities that are most important for your project. You might have something like: "build confidence in the software correctness," "demonstrate adequate usability." etc. You won't list all of the qualities, just those that are most important. —SS]

[You should also list the objectives that are out of scope. You don't have the resources to do everything, so what will you be leaving out. For instance, if you are not going to verify the quality of usability, state this. It is also worthwhile to justify why the objectives are left out. —SS]

[The objectives are important because they highlight that you are aware of limitations in your resources for verification and validation. You can't do everything, so what are you going to prioritize? As an example, if your system depends on an external library, you can explicitly state that you will assume that external library has already been verified by its implementation team. —SS]

2.3 Challenge Level and Extras

[State the challenge level (advanced, general, basic) for your project. Your challenge level should exactly match what is included in your problem statement. This should be the challenge level agreed on between you and the course instructor. You can use a pull request to update your challenge level (in TeamComposition.csv or Repos.csv) if your plan changes as a result of the VnV planning exercise. —SS]

[Summarize the extras (if any) that were tackled by this project. Extras can include usability testing, code walkthroughs, user documentation, formal proof, GenderMag personas, Design Thinking, etc. Extras should have already been approved by the course instructor as included in your problem statement. You can use a pull request to update your extras (in TeamComposition.csv or Repos.csv) if your plan changes as a result of the VnV planning exercise. —SS]

2.4 Relevant Documentation

[Reference relevant documentation. This will definitely include your SRS and your other project documents (design documents, like MG, MIS, etc). You can include these even before they are written, since by the time the project is done, they will be written. You can create BibTeX entries for your documents and within those entries include a hyperlink to the documents. —SS]

Author (2019)

[Don't just list the other documents. You should explain why they are relevant and how they relate to your VnV efforts. —SS]

3 Plan

[Introduce this section. You can provide a roadmap of the sections to come. —SS]

3.1 Verification and Validation Team

There will be a total of 7 distinct roles when it comes to our verification and validation team. These roles will be split amongst team members and members may share the same secondary roles to facilitate a broader range of tests without overwhelming one person. This division allows each team member to specialize in a certain aspect of our app when it comes to verification and validation. Attached below is a list of roles available in our VnV group along with the descriptions of what each role entails.

VnV Roles

- Test Case Designer
  - Develops detailed test cases for functional requirements, focusing on different aspects such as image quality, data parsing accuracy, and categorization correctness.
  - Works closely with the other developers to understand key components of the app and to ensure that test cases cover all scenarios, including edge cases.
- Quality Assurance Engineer
  - Ensures overall quality of the app by performing functional, integration, and system tests.
  - Coordinates end-to-end testing for receipt scanning and expense categorization workflows.
  - Collaborates with the Test Case Designer to verify that all test cases have been executed and outcomes meet expected results.
- Ai Model Validator
  - Focuses on testing the AI model specifically, evaluating its accuracy in parsing text and categorizing expenses.
  - Conducts model performance assessments under various conditions, such as diverse receipt layouts, lighting conditions, and languages if applicable.
  - Continuously monitors for model drift and helps recalibrate the model if accuracy degrades over time.

- Usability Tester
  - Evaluates the app's user interface and user experience for intuitiveness and ease of use.
  - Conducts usability testing sessions with other students or users in the target demographic (e.g., university students).
  - Provides feedback on the app's functionality and ensures that user feedback is incorporated into iterative improvements.
- Data Integrity Specialist
  - Ensures the accuracy and security of the data collected, especially sensitive information such as financial data on receipts.
  - Verifies that the app correctly anonymizes data if required and that categorization matches expected outputs based on provided datasets.
  - Works closely with the AI Model Validator to confirm data handling complies with privacy regulations and standards.
- Automation Engineer
  - Develops automated testing scripts to streamline regression testing and ensure the app functions consistently across different updates.
  - Automates repetitive tests, especially those related to scanning, categorization, and UI testing, to save time during development sprints.
  - Coordinates with the Test Case Designer to convert key test cases into automated tests for efficient reuse.
- Performance Tester
  - Measures and optimizes the app's performance, focusing on response time, load time, and battery usage (important for a mobile app).
  - Conducts stress tests to determine the app's behavior under heavy usage (e.g., scanning multiple receipts in a short period).

- Collaborates with developers to troubleshoot and resolve performance bottlenecks.

The following section delegates the roles above to each team member. Note that each team member has a secondary role which may be repeated amongst other members. The secondary role is to provide additional assistance when it comes to verification and validation and also serves to bring additional confidence that the system is working as it should.

Delegation of VnV Roles

Team Member 1

- Role(s)
  1. Test Case Designer
  2. Quality Assurance Engineer
- Responsibilities
  1. Creates comprehensive test cases for functional requirements.
  2. Executes tests to ensure all functionalities are covered.

Team Member 2

- Role(s)
  1. AI Model Validator
  2. Data Integrity Specialist
- Responsibilities
  1. Tests the AI model's accuracy in parsing receipts.
  2. Ensures data handling complies with security and privacy standards.

Team Member 3

- Role(s)
    1. Usability Tester
    2. Quality Assurance Engineer
- Responsibilities
    1. Conducts usability tests and collects user feedback.
    2. Supports quality assurance by testing overall workflows and integration.

Team Member 4

- Role(s)
    1. Automation Engineer
    2. Performance Tester
- Responsibilities
    1. Develops automated test scripts for repetitive testing.
    2. Measures app performance under different conditions to optimize speed and efficiency.

Team Member 5

- Role(s)
    1. Quality Assurance Engineer
    2. Performance Tester
- Responsibilities
    1. Coordinates end-to-end testing and verifies the app's functionality.
    2. Focuses on maintaining consistent performance across updates.

[Your teammates. Maybe your supervisor. You should do more than list names. You should say what each person's role is for the project's verification. A table is a good way to summarize this information. —SS]

3.2 SRS Verification Plan

[List any approaches you intend to use for SRS verification. This may include ad hoc feedback from reviewers, like your classmates (like your primary reviewer), or you may plan for something more rigorous/systematic. —SS]

[If you have a supervisor for the project, you shouldn't just say they will read over the SRS. You should explain your structured approach to the review. Will you have a meeting? What will you present? What questions will you ask? Will you give them instructions for a task-based inspection? Will you use your issue tracker? —SS]

[Maybe create an SRS checklist? —SS]

3.2.1 Approaches for SRS Verifiation

- Peer Review Feedback:
  - Each team member will review the SRS individually and provide ad hoc feedback. They will focus on identifying ambiguous language, incomplete requirements, and inconsistencies.
  - Each reviewer will log feedback in a shared document, categorizing issues by priority (e.g., high, medium, low).
- Primary Reviewer Session:
  - A primary reviewer from the team, ideally someone with experience in requirements verification, will perform a detailed examination. They will document findings, focusing on critical areas such as requirement clarity, feasibility, and completeness.
- External Peer Feedback:
  - If possible, solicit feedback from classmates or others familiar with requirements engineering. They can offer objective insights and identify issues that may have been overlooked internally.

3.2.2 Structured Internal Review Process

- Initial Team Meeting:
  - Hold a structured review meeting where team members present and discuss major sections of the SRS. Key points of focus should include functional requirements, assumptions, and constraints.
  - Prepare a set of questions to guide the discussion, such as:
    - "Are the requirements unambiguous and complete?"
    - "Is each requirement feasible and realistic?"
    - "Do the requirements adequately represent the user's needs?"
    - "Are there any areas that might be challenging to test or verify?"
- Task-Based Inspection:
  - Team members can take turns as "inspectors," responsible for a task-based evaluation of key areas:
    - Ensuring requirements align with project goals.
    - Verifying each requirement's testability and clarity.
    - Checking that the document is organized and easy to navigate.
- Issue Tracker:
  - Use an issue tracker or a collaborative tool to manage findings and resolutions. Each issue should include:
    - A summary of the issue.
    - Priority level (e.g., high, medium, low).
    - Suggested resolution steps.
    - A status field for tracking progress until resolved.

3.2.3 SRS Quality Checklist for Verification

Checklist Item	**Description**	**Pass/Fail**
Requirements Clarity	All requirements are clear, unambiguous, and written in active voice.	
Completeness	The SRS includes all functional and non-functional requirements, assumptions, and constraints.	
Consistency	Requirements are consistent across the document, with no contradictory statements.	
Testability	Each requirement is formulated so that it can be tested through measurable criteria.	
Feasibility	All requirements are realistic and achievable within the project's scope and resources.	
Traceability	Each requirement is traceable to a higher-level objective or user need.	
Maintainability	The document is well-organized, with clear headings and numbered requirements for easy referencing.	
Priority and Dependencies	Requirements are prioritized, and dependencies are explicitly stated where applicable.	
User-Centered Language	Requirements reflect user needs and use terminology consistent with user and stakeholder language.	
Compliance with Standards	The SRS complies with any applicable standards or best practices in software requirements engineering.	

3.3 Design Verification Plan

[Plans for design verification —SS]

[The review will include reviews by your classmates —SS]

[Create a checklists? —SS]

3.3.1 Specification Verification

Checklist Items	**Description**	**Verification Method**	**Pass /Fail**	**Comments**
Requirements Completeness	Ensure all system and user requirements are addressed in the design.	Requirements Traceability Matrix		
System Requirements Compliance	Verify that design conforms to software/system specifications.	Specification Review		
Compliance with Standards	Ensure compliance with coding standards, industry best practices, and regulatory guidelines.	Code Review, Standards Checklist		
Security Requirements	Confirm that the security design follows the requirements, especially for data privacy and user information.	Security Audit		

3.3.2 Functional Verification

Checklist Items	**Description**	**Verification Method**	**Pass /Fail**	**Comments**
Receipt Scanning Capability	Verify that receipt scanning feature works as expected with various image qualities.	Functional Testing		
Expense Categorization Accuracy	Confirm that expenses are correctly categorized by the AI model.	Model Accuracy Testing		
User Interface (UI) Responsiveness	Ensure that UI elements are responsive across devices and screen sizes.	UI Testing		
Error Handling and Recovery	Verify that the app gracefully handles errors, such as scanning failures or incorrect categorizations.	Error Simulation		
Integration with External Services	Confirm integration with any third-party services (e.g., cloud storage or payment platforms).	Integration Testing		

3.3.3 Performance Validation

Checklist Items	**Description**	**Verification Method**	**Pass /Fail**	**Comments**
System Performance Under Load	Test app performance with simultaneous receipt scans to assess system stability and responsiveness.	Load Testing		
AI Model Processing Time	Measure the average processing time for receipt categorization.	Model Timing Analysis		
Battery and Resource Consumption	Verify that app resource usage is within acceptable limits to preserve device performance and battery life.	Resource Monitoring		
Network Efficiency	Ensure the app handles network limitations (e.g., slow or intermittent connections) without data loss.	Network Simulation		
Latency for Real-Time Features	Measure latency in real-time features, ensuring a smooth user experience.	Latency Testing		

3.3.4 Document Validation

Checklist Items	**Description**	**Verification Method**	**Pass /Fail**	**Comments**
User Documentation Completeness	Verify that user manuals, installation guides, and support documentation are complete and accurate.	Documentation Review		
Developer Documentation Completeness	Ensure all developer-focused documentation (e.g., API docs, setup instructions) is detailed and up to date.	Documentation Review		
Code Documentation	Confirm that all code modules and functions are documented according to guidelines.	Code Review		
Test Plans and Results	Verify the completeness of test plans and that test results meet required standards.	Test Report Review		
Change Log and Version History	Ensure that the change log and version history are comprehensive and well-documented.	Change Log Review		

3.4 Verification and Validation Plan Verification Plan

[The verification and validation plan is an artifact that should also be verified. Techniques for this include review and mutation testing. —SS]

[The review will include reviews by your classmates —SS]

[Create a checklists? —SS]

3.5 Implementation Verification Plan

[You should at least point to the tests listed in this document and the unit testing plan. —SS]

[In this section you would also give any details of any plans for static verification of the implementation. Potential techniques include code walkthroughs, code inspection, static analyzers, etc. —SS]

[The final class presentation in CAS 741 could be used as a code walkthrough. There is also a possibility of using the final presentation (in CAS741) for a partial usability survey. —SS]

3.6 Automated Testing and Verification Tools

[What tools are you using for automated testing. Likely a unit testing framework and maybe a profiling tool, like ValGrind. Other possible tools include a static analyzer, make, continuous integration tools, test coverage tools, etc. Explain your plans for summarizing code coverage metrics. Linters are another important class of tools. For the programming language you select, you should look at the available linters. There may also be tools that verify that coding standards have been respected, like flake9 for Python. —SS]

[If you have already done this in the development plan, you can point to that document. —SS]

[The details of this section will likely evolve as you get closer to the implementation. —SS]

1. **Unit Testing Frameworks**

    - **Jest**: Jest is the default testing framework for React Native applications. It supports TypeScript, has a rich API for writing unit tests, and comes with built-in mocking capabilities.
    - **React Testing Library**: This library works well with Jest for testing React components, focusing on user interactions and component behavior rather than implementation details.

2. **Static Analysis Tools**

    - **ESLint**: ESLint is a powerful linter for JavaScript and TypeScript. It can be configured with TypeScript support to enforce coding standards, detect potential errors, and maintain code quality. It can also be extended with plugins for React and serves as industry best practices for JavaScript/TypeScript related-projects.
    - **Prettier**: Integrating Prettier alongside ESLint helps ensure consistent code formatting across the codebase, which helps in improving readability.

3. **Continuous Integration (CI) Tools**

    - **GitHub Actions**: This tool can be set up to run workflows that execute tests, linters, and builds automatically on every pull request or commit, ensuring that code quality is maintained consistently (i.e. Husky).

- **Jenkins**: Jenkins is an open-source automation server that facilitates continuous integration and continuous deployment processes. It can be configured to automatically run tests, linting, and builds every time code is pushed to the repository, ensuring that code quality is maintained consistently.

4. **Test Coverage Tools**

   - **Istanbul (nyc)**: Istanbul is a code coverage tool that integrates with Jest, providing detailed reports on which parts of the codebase are covered by tests, helping to identify untested code.
   - **Codecov or Coveralls**: These services can be used to visualize and summarize code coverage metrics after each build, providing insights into overall coverage and trends over time.

5. **Profiling Tools**

   - **React Native Performance Monitor:**: This built-in tool in React Native helps track performance metrics such as frame rates and memory usage, providing insights into the app's performance.
   - **Flipper**: A platform for debugging mobile apps, Flipper offers a variety of plugins that assist in profiling and monitoring app performance.

6. **Mutation Testing**

   - **Stryker**: Stryker is a mutation testing framework for JavaScript and TypeScript. It can help assess the effectiveness of tests by introducing small changes in the code (mutations) and checking if the tests can catch them.

7. **Plans for Summarizing Code Coverage Metrics**

   - **Daily/Weekly Reports:**: The CI pipeline can be configured to generate code coverage reports with every build and aggregate these reports weekly. Tools like Codecov can be used to visualize trends over time.
   - **Review Meetings**: Regular review meetings can be conducted to discuss coverage metrics with the team, identifying areas needing improvement and setting goals for coverage percentages.

- **Integrate Coverage Reports in PRs**: Coverage reports should be included in pull request reviews, ensuring that any code changes are evaluated in the context of their impact on overall coverage.

For additional information, please consult the development plan.

3.7 Software Validation Plan

[If there is any external data that can be used for validation, you should point to it here. If there are no plans for validation, you should state that here. —SS]

[You might want to use review sessions with the stakeholder to check that the requirements document captures the right requirements. Maybe task based inspection? —SS]

[For those capstone teams with an external supervisor, the Rev 0 demo should be used as an opportunity to validate the requirements. You should plan on demonstrating your project to your supervisor shortly after the scheduled Rev 0 demo. The feedback from your supervisor will be very useful for improving your project. —SS]

[For teams without an external supervisor, user testing can serve the same purpose as a Rev 0 demo for the supervisor. —SS]

[This section might reference back to the SRS verification section. —SS]

4 System Tests

[There should be text between all headings, even if it is just a roadmap of the contents of the subsections. —SS]

4.1 Tests for Functional Requirements

[Subsets of the tests may be in related, so this section is divided into different areas. If there are no identifiable subsets for the tests, this level of document structure can be removed. —SS]

[Include a blurb here to explain why the subsections below cover the requirements. References to the SRS would be good here. —SS]

4.1.1 Area of Testing1

[It would be nice to have a blurb here to explain why the subsections below cover the requirements. References to the SRS would be good here. If a section covers tests for input constraints, you should reference the data constraints table in the SRS. —SS]

Title for Test

1. test-id1

   Control: Manual versus Automatic

   Initial State:

   Input:

   Output: [The expected result for the given inputs. Output is not how you are going to return the results of the test. The output is the expected result. —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

2. test-id2

   Control: Manual versus Automatic

   Initial State:

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

4.1.2 Area of Testing2

...

4.2 Tests for Nonfunctional Requirements

[The nonfunctional requirements for accuracy will likely just reference the appropriate functional tests from above. The test cases should mention reporting the relative error for these tests. Not all projects will necessarily have nonfunctional requirements related to accuracy. —SS]

[For some nonfunctional tests, you won't be setting a target threshold for passing the test, but rather describing the experiment you will do to measure the quality for different inputs. For instance, you could measure speed versus the problem size. The output of the test isn't pass/fail, but rather a summary table or graph. —SS]

[Tests related to usability could include conducting a usability test and survey. The survey will be in the Appendix. —SS]

[Static tests, review, inspections, and walkthroughs, will not follow the format for the tests given below. —SS]

[If you introduce static tests in your plan, you need to provide details. How will they be done? In cases like code (or document) walkthroughs, who will be involved? Be specific. —SS]

4.2.1 Area of Testing1

Title for Test

1. test-id1

   Type: Functional, Dynamic, Manual, Static etc.

   Initial State:

   Input/Condition:

   Output/Result:

   How test will be performed:

2. test-id2

   Type: Functional, Dynamic, Manual, Static etc.

   Initial State:

   Input:

Output:

How test will be performed:

4.2.2 Area of Testing2

...

4.3 Traceability Between Test Cases and Requirements

[Provide a table that shows which test cases are supporting which requirements. —SS]

5 Unit Test Description

[This section should not be filled in until after the MIS (detailed design document) has been completed. —SS]

[Reference your MIS (detailed design document) and explain your overall philosophy for test case selection. —SS]

[To save space and time, it may be an option to provide less detail in this section. For the unit tests you can potentially layout your testing strategy here. That is, you can explain how tests will be selected for each module. For instance, your test building approach could be test cases for each access program, including one test for normal behaviour and as many tests as needed for edge cases. Rather than create the details of the input and output here, you could point to the unit testing code. For this to work, you code needs to be well-documented, with meaningful names for all of the tests. —SS]

5.1 Unit Testing Scope

[What modules are outside of the scope. If there are modules that are developed by someone else, then you would say here if you aren't planning on verifying them. There may also be modules that are part of your software, but have a lower priority for verification than others. If this is the case, explain your rationale for the ranking of module importance. —SS]

5.2 Tests for Functional Requirements

[Most of the verification will be through automated unit testing. If appropriate specific modules can be verified by a non-testing based technique. That can also be documented in this section. —SS]

5.2.1 Module 1

[Include a blurb here to explain why the subsections below cover the module. References to the MIS would be good. You will want tests from a black box perspective and from a white box perspective. Explain to the reader how the tests were selected. —SS]

1. test-id1

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

2. test-id2

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

3. ...

5.2.2 Module 2

...

5.3 Tests for Nonfunctional Requirements

[If there is a module that needs to be independently assessed for performance, those test cases can go here. In some projects, planning for nonfunctional tests of units will not be that relevant. —SS]

[These tests may involve collecting performance data from previously mentioned functional tests. —SS]

5.3.1 Module ?

1. test-id1

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input/Condition:

   Output/Result:

   How test will be performed:

2. test-id2

   Type: Functional, Dynamic, Manual, Static etc.

   Initial State:

   Input:

   Output:

   How test will be performed:

5.3.2 Module ?

...

5.4 Traceability Between Test Cases and Modules

[Provide evidence that all of the modules have been considered. —SS]

References

Author Author. System requirements specification. https://github.com/..., 2019.

6 Appendix

This is where you can place additional information.

6.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

6.2 Usability Survey Questions?

[This is a section that would be appropriate for some projects. —SS]

Appendix — Reflection

[This section is not required for CAS 741 —SS]

The information in this section will be used to evaluate the team members on the graduate attribute of Lifelong Learning.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?

2. What pain points did you experience during this deliverable, and how did you resolve them?

3. What knowledge and skills will the team collectively need to acquire to successfully complete the verification and validation of your project? Examples of possible knowledge and skills include dynamic testing knowledge, static testing knowledge, specific tool usage, Valgrind etc. You should look to identify at least one item for each team member.

4. For each of the knowledge areas and skills identified in the previous question, what are at least two approaches to acquiring the knowledge or mastering the skill? Of the identified approaches, which will each team member pursue, and why did they make this choice?