System Verification and Validation Plan for Flow

Team 9, min-cut
Ethan Patterson
Hussain Muhammed
Jeffrey Doan
Kevin Zhu
Chengze Zhao

October 28, 2025

Revision History

Date	Version	Notes
Date 1	1.0	Notes
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

1 Symbols, Abbreviations, and Acronyms **iv**

2 General Information **1**

2.1 Summary 1

2.2 Objectives 1

2.3 Extras 1

2.4 Relevant Documentation 2

3 Plan **2**

3.1 Verification and Validation Team 2

3.2 SRS Verification 2

3.3 Design Verification 3

3.4 Verification and Validation Plan Verification 3

3.5 Implementation Verification 3

3.6 Automated Testing and Verification Tools 5

3.7 Software Validation 6

4 System Tests **7**

4.1 Tests for Functional Requirements 7

4.1.1 Area of Testing1 — Editing and Canvas Operations . . 7

4.1.2 Area of Testing2 — File Management and Persistence . 9

4.2 Tests for Nonfunctional Requirements 10

4.2.1 Area of Testing1 11

4.2.2 Area of Testing2 11

4.3 Traceability Between Test Cases and Requirements 11

5 Unit Test Description **13**

5.1 Unit Testing Scope 13

5.2 Tests for Functional Requirements 13

5.2.1 Module 1 13

5.2.2 Module 2 14

5.3 Tests for Nonfunctional Requirements 15

5.3.1 Module ? 15

5.3.2 Module ? 15

5.4 Traceability Between Test Cases and Modules 16

6 Appendix **17**

6.1 Symbolic Parameters 17

6.2 Usability Survey Questions? 17

List of Tables

1 Traceability Matrix for Functional Requirements 12

[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
PR	Pull request

[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 Extras

[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

[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

[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]

3.3 Design Verification

[Plans for design verification —SS]

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

[Create a checklists? —SS]

3.4 Verification and Validation Plan Verification

[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

[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]

Firstly, the tests outlined in the System Tests (and eventually the Unit Test Description) section will be used for verification. The purpose of these tests are to ensure the system behaves as expected given the specification, which naturally assists in verification of the implemented system. It is expected that not every individual unit test will be specified in the unit testing plan and devs should not withhold from writing a useful unit test just because it is not specified in this document. An example is writing a new unit test to cover the expected functionality of a newly discovered bug. Devs are expected to write tests that cover their code at the specified coverage rate of *TEST_COVERAGE_RATE*.

Beyond the specified test cases, the product will be test driven by members of the team for taking notes during lectures. All members of the team are expected to know the general expected functionality of the system, so having the team test the product as a user would greatly benefits verifying the product. Additionally, this is a feasible plan given the time frame and resources of the project. All team members attend classes on a daily basis and have the opportunity to use Flow during their lecture time.

While manual and automated dynamic tests are essential for verification of the product, non-dynamic testing approaches can also be extremely effective and be used throughout the development process. Typical static testing approaches include static code analyzers, code walkthroughs, and code review. These three approaches will be used by the team as part of the verification process.

A linting tool will be selected and each team member will be required to use it when developing locally. To enforce its use, a new workflow step will be added to the GitHub actions that will run the linting tool on an opened pull request.

Code review is expected to conducted on every pull request before it can be approved and merged. To keep the plan realistic, only one review will be required per PR, but more are encouraged if possible. In the case of looming hard deadlines that must be met in a timely manner, the team may come to a collective decision to skip a rigorous review process on a PR if they deem the change to be small enough. However, the GitHub actions pipeline is still expected to be run and pass before merging to main.

Lastly, synchronous code walkthrough may be conducted per a dev's request. This is expected to supplement a code review if a reviewer requests it. It may also be proposed by the code owner if they feel knowledge sharing and familiarity of the code/architecture will be beneficial for the team. Examples include a new module that will be reused by other devs for other parts of the codebase or some critical functionality that would benefit from group review/verification. Since there is more overhead with this approach due to there needing to be a scheduled time, this is more of an option that is open to the team rather than a strictly enforced testing plan approach.

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]

Firstly, the planned tech stack for the project is an Electron app with a React frontend using TypeScript.

For automated unit testing, we will use the Jest framework, which is a JavaScript testing framework that can be used with TypeScript and React. For more end-to-end feature testing, we will use Playwright, which is built for end-to-end testing web apps, but also has support for Electron apps.

For profiling and debugging, since Electron is built on top of Chromium, we can use Chrome DevTools to measure the performance and memory usage of our app.

For static analysis and linting we will use ESLint with the TypeScript and

React plugins, which will enforce coding rules and find anti-patterns. Additionally, Prettier will be used for formatting. The combination of these tools will lead to a higher quality codebase with consistent styling.

Pnpm will be our build system. We can define pnpm scripts for running, building, linting, and testing our app.

GitHub Actions will be used for continuous integration. The pipeline will compile the app, run the automated tests, linting step, and formatting step on each PR and main merge.

Jest has a built in feature for measuring and summarizing code test coverage, simply by passing the '–coverage' flag. This report will then be integrated into the CI pipeline through Coveralls, which is free for public GitHub repos and has built in GitHub integration.

3.7 Software Validation

[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]

There are no plans for formal validation of the project as there is no formal individual stakeholder/supervisor for the project.

As a team we are eliciting requirements amongst ourselves and we are among

the projected users of Flow, so we expect to work as a group throughout the development process to tweak existing requirements and add missing requirements as they are discovered.

4 System Tests

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

This section defines the system-level tests that will verify the correctness of the system's functional requirements as specified in the SRS (Section S.2). Each test case focuses on validating user-facing behaviors and ensuring that the system performs as expected under normal and boundary conditions.

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]

The following subsections group tests into functional areas that align with the SRS: 1. **Editing and Canvas Operations** – covering note creation, editing, and manipulation (SRS F211–F216). 2. **File Management and Persistence** – covering saving, loading, and export functions (SRS F217–F243).

4.1.1 Area of Testing1 — Editing and Canvas Operations

[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]

This area tests core editing functions that define the user experience — cre-

ating, modifying, and visually managing elements on the canvas. Successful execution validates that the system's interactive core behaves as specified in the SRS functional requirements F211–F216.

Note Creation and Editing

1. test-F211

   Control: Automatic.

   Initial State: Application launched with no open document.

   Input: User creates a new note and types text content.

   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] A new note file is created and displayed in the editor window. Typed content appears immediately in the canvas with no delay.

   Test Case Derivation: [Justify the expected value given in the Output field —SS] The expected outcome is derived from F211 in the SRS, which specifies that users can create and edit notes in real time. The test verifies that note objects are correctly instantiated and rendered without performance issues.

   How test will be performed: Automated UI testing via simulated input sequence that triggers note creation, typing, and real-time rendering validation.

2. test-F214

   Control: Automatic

   Initial State: Canvas open with editing mode enabled.

   Input: User inserts a shape (rectangle) and moves it across the canvas.

   Output: [The expected result for the given inputs —SS] The shape appears correctly and follows the user's movement without lag or distortion.

Test Case Derivation: [Justify the expected value given in the Output field —SS] Expected behavior is defined by F214 and F216, requiring that the user can freely reposition and resize elements. The expected value is that the object's final position and dimensions match those in the test specification.

How test will be performed: Automated functional test using simulated mouse actions and coordinate comparison of expected versus rendered object placement.

4.1.2 Area of Testing2 — File Management and Persistence

This area validates system behavior related to file operations such as saving, loading, and exporting notes. It ensures compliance with SRS functional requirements F217–F243, verifying that user data is properly stored and retrieved.

Saving and Loading Notes

1. test-F217

   Control: Automatic

   Initial State: A note with unsaved text and shape content exists in the editor.

   Input: User selects "Save" and then "Open" from the menu.

   Output: The note reopens with all previously entered text and shapes intact.

   Test Case Derivation: Expected results come from SRS F217 and F241, which require data persistence and accurate file recovery. The test verifies successful serialization and deserialization of notes.

   How test will be performed: Automated file I/O comparison between pre-save and post-load content to confirm identical state restoration.

2. test-F243

Control: Manual

Initial State: A completed note is open in the editor.

Input: User selects “Export as PDF.”

Output: A PDF file is generated containing the correct layout, formatting, and graphical content.

Test Case Derivation: Derived from F243, which specifies export functionality for interoperability. Expected outcome is that the exported file visually matches the on-screen content with no missing or altered elements.

How test will be performed: Manual verification by comparing the exported PDF output to the in-application rendering.

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]

Table 1: Traceability Matrix for Functional Requirements

Requirement ID	**Requirement Description**	**Related Test(s)**
F211	The system shall allow users to create and edit notes.	Test-F211: Verify that users can successfully create new notes and edit existing ones.
F212	The system shall allow users to organize notes into folders and subfolders.	Test-F212: Verify that note organization functions correctly, allowing drag-and-drop or selection into designated folders.
F217	The system shall allow users to view and pan across different notes and sections within the workspace.	Test-F217: Verify panning functionality and proper rendering of note sections during navigation.
F241	The system shall support file management operations (save, rename, delete).	Test-F241: Confirm file management operations perform as expected, including proper updates to the note directory.
F242	The system shall autosave changes at regular intervals (smaller than or equal to 60 seconds).	Test-F242: Check that autosave triggers correctly and saves the latest note content within the defined time frame.

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.

TEST_COVERAGE_RATE - 70%. Ideally this value would be 100%, but 70% was chosen as a realistic balance between time constraints and verification quality.

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?
    - Chengze -
    - Ethan -
    - Hussain -
    - Jeffrey -
    - Kevin -
2. What pain points did you experience during this deliverable, and how did you resolve them?
    - Chengze -
    - Ethan -
    - Hussain -
    - Jeffrey -

- Kevin -

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.

   Collectively, the team will need to learn a variety of skills to complete the verification process. Five new skills include building an automated testing pipeline with GitHub actions, writing unit tests for frontend components, writing end-to-end tests, integrating code coverage metrics, and performance testing/verification. Firstly, while some of the team is familiar with CI/CD pipelines, none of us have created and maintained one ourselves before. We have already tweaked our GitHub workflows for managing the building for our latex and adoc files, but setting up the pipeline for more complex steps like testing, linting, etc will require more learning. Secondly, while we all have experience writing unit tests in some form, writing tests for React and component testing will be a new concept. Additionally, writing end-to-end tests and setting up the infrastructure for running them will also be new for most of the team. Both of these testing approaches will involve learning to use Jest and Playwright, the testing frameworks we have decided to use. Some of us also have experience with using Coveralls for code coverage metrics, but again we will need to learn how to set it up ourselves. Lastly, performance testing to verify nonfunctional requirements is another new concept for most of the team, and using Chrome Dev Tools will be required to complete that.

   Generally, the team will need to collectively learn more about and improve test writing skills. This includes writing better software that is testable, which has not always been a focus in our course work. Developing the habit of writing tests to cover the code you write is important and will be new for some of us.

   As mentioned, validation is not as much of a concern for this project.

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?

- Chengze -
- Ethan -
- Hussain -
- Jeffrey -
- Kevin -