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

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
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 Verification and Validation (VnV) plan outlines the strategies that will be employed to ensure that Flow meets its specified requirements and functions correctly. This document covers the methods and criteria used to confirm that the software is built in a way that satisfies the outlined requirements, and that fulfills its intended purpose.

2 General Information

2.1 Summary

Flow is a note-taking application designed to convert keyboard input into visual diagrams and structured graphical representations. In addition to basic note-taking software features, such as creating, opening, editing, saving, and deleting notes, Flow aims to enhance user productivity by allowing users to quickly generate visual content from text input.

Each note will be represented as a scrollable canvas where users can insert, edit, or delete text within text boxes. Flow will also support the insertion of predefined geometric shapes, as well as custom workflow shapes that users can define. The Verification and Validation (VnV) process will focus on ensuring that these core features function as specified, user interface elements work efficiently and intuitively, and that user interactions meet the outlined requirements. noindent

2.2 Objectives

The primary objective of this Verification and Validation (VnV) plan is to ensure that Flow meets the specified requirements outlined in the Software Requirements Specification (SRS) document. This process aims to build confidence that the system functions correctly, is reliable, and provides a user-friendly experience. The objectives also aims to test activities that mitigate the risks identified in the Hazard Analysis document. This will ensure that the system is robust and can handle potential failure scenarios gracefully.

Specifically, the VnV plan focuses on the following objectives:

- **Document Management Correctness**: Verifying that file related operations, such as creating, opening, saving, deleting and file organization of notes, function as intended without data loss or corruption.
- **Note and Diagram Manipulation Correctness**: Ensuring that editing features related to note-taking and diagram creation work as specified. This includes textbox and geometric diagram manipulation (inserting, resizing, moving, and deleting), as well as text formatting capabilities (bolding, italicizing, underlining, etc).
- **Data Integrity**: Confirming that all user inputs and modifications are saved accurately and can be retrieved without corruption or loss.
- **Performance and Responisiveness**: Verifying that the user interactions occur within acceptable time frames, ensuring a smooth and efficient user experience.
- **System reliability**: Confirming that the system operates consistently in different sessions without crashes or unexpected behavior.

Objectives that are out of scope for this VnV plan include:

- **Formal Usability Testing**: Due to resource constraints, formal usability testing with a large user base will not be conducted. Instead, an informal usability test from team members within the development team will be used as these members fall into the target user demographic.
- **Security Testing**: Given the nature of Flow being a local note-taking noindent application, security testing is not prioritized in this VnV plan. The focus will be on functionality and usability rather than security vulnerabilities.
- **External Library Verification**: Any third-party libraries or frameworks used for rendering or GUI components will be assumed to have been verified by their respective development teams.

2.3 Extras

As part of the VnV prcess for Flow, the project will include the following two approved extras: Usability Testing and a User Manual.

- **Usability Testing**: An informal, but structured test will be conducted with team members who fall within the target user demographic. It will be done to evaluate the intuitiveness and efficiency of the user interface and workflows of Flow. Methods will be made as systematic and objective as possible, to help closely identify areas for improvement. Participants will complete a set of predefined tasks (e.g. creating and organizing notes, inserting and editing diagrams, etc.) while their performance and completion times are recorded (although participants are team members, they will not be involved in the elicitation of the specific details of the set tasks to avoid bias). Afterwards, participants will be given an opportunity to provide feedback on their experience, including any difficulties encountered and suggestions for enhancements.
- **User Manual**: A comprehensive user manual will be created to guide new users through the features and functionalities of Flow. The manual will include step-by-step instructions for creating, editing, and managing notes and diagrams. It will also have explanations of various tools and keyboard shortcuts. The user manual will serve as both a user support resource and a validation tool to ensure that all features are accessible and understandable to users.

2.4 Relevant Documentation

The following documents are relevant and closely linked to the Verification and Validation (VnV) document. These documents define and guide the development of Flow:

- **Software Requirements Specification (SRS)**: The SRS establishes the functional and non-functional requirements for Flow, forming the foundation for the VnV activities. Sections such as the Functional

Overview (G.4), High-Level Usage Scenarios (G.5), and System Components (S.1) were referenced to ensure the testing objectives and procedures align with Flow's specified features and structure.

- **Problem Statement**: The Problem Statement outlines the project's goals and motivations. It was referenced to ensure that the VnV objectives align with the intended purpose of Flow. The Extras section of the Problem Statement was also used to identify the approved extras for the project, which are included in this VnV plan.

- **Design Documents (MG, MIS)**: The Module Guide (MG) and Module Interface Specification (MIS) will provide detailed descriptions of the system's architecture and module interactions. During develpoment phases of the project, as well as development of unit and integration testing, the MG and MIS will be referenced to ensure that each component performs as intended and that the interactions between modules function correctly.

- **Hazard Analysis**: The Hazard Analysis document identifies potential risks and failure areas within Flow. This document was referenced to inform the VnV plan's objectives. Key risks and such as data loss, rendering lag, and unreliability were identified and considered. This aims to ensure that testing activities address these risks and validate the system's robustness against potential hazards.

3 Plan

This section will go over the Verification and Validation Team members and the techniques and methodology required for a VnV Plan's success.

3.1 Verification and Validation Team

1. **Ethan** — Team Lead / Test Manager
    - Develop test plan, testing strategies, and test timelines
2. **Hussain** — Test Analyst

- Create and review detailed tests for functional, non-functional, and edge cases required for the project

3. **Oliver** — Quality Assurance Tester
    - Conduct tests to verify that all features work as expected according to the SRS
4. **Jeffrey** — Automation Tester
    - Design, develop, and maintain most of the automated test scripts used for repetitive and regression testing
5. **Kevin** — Document Specialist
    - Write and maintain most of the documentation, referring to any test plans, test cases, reports, or defect logs

3.2 SRS Verification

1. Internal Review and Cross-Check
    (a) This is the first level of verification for the SRS. The team will conduct a review as a whole, reviewing each member's contribution and giving feedback. Additionally, this will help members confirm consistency between each individually worked on part.
    (b) Moreover, the team will also be given the opportunity to cross-check with the requirements listed in any previous documents to ensure there is consistency between documentation.
2. Peer Review
    (a) This verification process will allow our team to seek assistance from a peer capstone team. This portion will allow another team to verify that the SRS is filled to completion and is error-free, ready for validation and development.
    (b) Additionally, this gives our team a chance to review another capstone's SRS, which will allow us to cross-reference with our SRS, verifying that both documents adhere to the Myers SRS convention and include every necessary portion.

3. Check List Review

    (a) This level of verification will have the team review and cross-reference the SRS checklist once again after the deliverable has been completed.

    (b) This is an extensive SRS checklist provided by our course instructor and is a professional way to ensure that our SRS contains everything required of us.

    (c) Going over this as a team once the document has been completed allows us to verify that all members adhered to the checklist when creating their portion of the SRS document.

3.3 Design Verification

1. Purpose:

    (a) We will use this to ensure that the design of our application meets the specifications and user needs outlined in the SRS. By referencing the SRS, we can verify that our application is designed as intended and aligns with any goals or statements made during the planning process.

2. Design Verification Scope:

    (a) This plan will cover the verification of the following components:

        i. User Interface (UI): Verify that the UI is intuitive and is minimized to accommodate a more keyboard-friendly environment, which would not require any mouse involvement.

        ii. Diagram Creation: Verify that the diagram creation feature works using only a keyboard and no mouse/trackpad. Ensure that diagrams are created with the same freedom and range as writing on a piece of paper or using a mouse.

        iii. Accessibility and Usability: Verify that users can efficiently use the application without the need for other external tools aside from a keyboard.

        iv. Performance: Verify that the system works as intended even when generating multiple objects or diagrams.

3. Verification Objectives:

    (a) Comply with Functional Requirements: Verify that the design supports all key features and complies with the functional requirements listed in all past documents, such as the SRS.

    (b) Verify Performance: Verify that the application can handle multiple diagrams and object elements.

4. Design Verification Methodology:

    (a) Verification through Design Reviews:

        i. Internal Design Review:

            A. The development team will conduct an internal review of the application to ensure that we are complying with all the functional requirements specified in the SRS and other related documents.

            B. The team will also verify that the application is user-friendly by using past knowledge on product usability and referencing existing note-taking applications.

        ii. Design Walkthrough

            A. The development team will conduct walkthroughs with other team members who may not have been involved in specific feature development. Additionally, peers and friends may be used to conduct a proper walkthrough session.

            B. During the walkthrough, ensure that users are able to navigate the application and create diagrams using the keyboard as intended.

    (b) Functional Testing:

        i. Test Case Development:

            A. Develop test cases that focus on verifying key design features, such as keyboard navigation or the creation of diagrams or objects using shortcuts.

    (c) Performance Testing:

        i. Load Testing:

A. Perform tests that will create strain on a system through the creation of multiple objects or diagrams that are being manipulated. Verify that the application is able to perform well under this load with the system requirements specified in the SRS.

ii. Speed Test:

A. Verify that all interactions (i.e., diagram/object creation and manipulation) are quick and responsive with minimal lag or delay.

5. Design Verification Criteria:

(a) Usability Criteria:

i. Users can create diagrams using only keyboard shortcuts.

ii. Users can navigate the user interface without relying on a mouse.

iii. No critical keyboard shortcut conflict, and functionality across all supported platforms.

(b) Functional Criteria:

i. All diagram creation features are functional. This includes adding shapes, manipulating elements, connecting existing objects, and deleting components, without the use of a mouse.

(c) Compliance with Requirements:

i. The design must meet all requirements outlined in the Software Requirements Specification (SRS).

3.4 Verification and Validation Plan Verification

1. Purpose:

(a) To verify that the VnV Plan is complete, correct, and meets the standards for verifying and validating our application.

2. VnV Plan Verification Scope:

(a) Completeness: Verify that all necessary sections and objectives are included.

(b) Clarity and Precision: Verify that the plan is clearly written and unambiguous.

(c) Feasibility: Verify that all techniques and processes described to be used in the verification and validation process are practical and achievable.

3. VnV Plan Verification Objectives:

   (a) Verify that the VnV Plan is comprehensive and covers all necessary portions required of a complete verification and validation plan.

   (b) Verify that the techniques outlined for verification are appropriate for validating the functional requirements listed in the SRS.

   (c) Verify that the plan can realistically be executed and will assist and result in the resolution of any defects found in our application.

4. Verification Techniques:

   (a) Internal Verification Plan Review:

      i. The entire capstone team will collectively review the document once finished. This is to verify that all members have completed each task to a standard that the team can agree on.

      ii. The team will also be utilizing the checklist provided by the course instructor. This will help in verifying that the VnV Plan is completed correctly and incorporates all the main features and sections required of a proper VnV Plan.

   (b) Peer Review:

      i. The VnV Plan will also be reviewed by another project team that has also been tasked to work on a similar VnV Plan for their capstone. This will help verify that our team has completed the document properly by comparing sections and content with another completed VnV document in the same cohort.

      ii. Moreover, while our team also reviews the partnered team's VnV Plan, we can cross-reference ours with theirs to verify that both teams have incorporated all the necessary sections and techniques required for this deliverable.

(c) Mutation Testing:

i. Our team will incorporate mutation testing to verify that our VnV Plan is working as intended. In order to do this, we will be incorporating small changes or "mutants" and testing to see if the verification and validation process noted would catch them. If the plan catches these changes, it will help confirm that the VnV plan is comprehensive and effective.

ii. Example of possible mutations:

A. Mutation 1: Remove a key verification step (e.g., Usability Verification) and see if the VnV process identifies this omission.

B. Mutation 2: Add an unnecessary or incorrect verification technique (e.g., object creation via mouse) and see if the VnV process can identify that the new technique is unsuitable or conflicts with other requirements.

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 — creating, 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]

The following section splits the tests into sections based on the Non-Functional requirements.:

1. **Performance.** - Covers the performance of the app while making and editing notes. (SRS NF211, NF218)

2. **Usability and Accessibility.** - Covers the users interactions with the note taking system. (SRS NF213-NF214, NF217)

3. **Consistency of Open and Closed Notes.** - Covers data not being lost or corrupted. (SRS NF212, NF215, NF216)

4.2.1 Area of Testing1 - Perfomance of Editing and Canvas Operations

Latency Test

1. test-NF211

   Type: Dynamic, Manual.

   Initial State: Application launched with an empty open note and a flag to show latency.

   Input/Condition: Adding shapes and text

   Output/Result: Shapes or text are shown on screen within *DISPLAY_LATENCY*. ms.

   How test will be performed: The tester will open a note and then attempt to add a shape and a text box, noting the latency for each.

2. test-NF218

   Type: Dynamic, Manual.

   Initial State: Application launched with a flag to show latency and an open note that contains *SHAPE_LOAD_TEST_COUNT*. shapes.

   Input/Condition: Adding shapes and text to the current note.

   Output/Result: Shapes or text are shown on screen within *DISPLAY_LATENCY*. ms.

   How test will be performed: The tester will open a pre-made note with *SHAPE_LOAD_TEST_COUNT*. shapes note and then attempt to add another shape, taking note of the latency.

4.2.2 Area of Testing2 - Usability and Accessibility

1. test-NF213

   Type: Dynamic, Manual.

   Initial State: Empty Linux, Windows, Mac computer.

   Input/Condition: Creating and opening note, adding shapes and text.

   Output/Result: Shapes or text are shown on screen

How test will be performed: The tester will open the program on various different operating systems to test if they function on each platform.

2. test-NF214

   Type: Dynamic, Manual.

   Initial State: Application launched with an open empty note.

   Input/Condition: Adding shapes and text via keyboard short cuts

   Output/Result: Shapes or text are shown on screen when short cuts are used.

   How test will be performed: The tester will open a note and then attempt to add a note and a shape and a text box via the corresponding short cut.

3. test-NF217

   Type: Dynamic, Manual, Usability Survey.

   Initial State: Application launched without a note.

   Input: Have and external user try to create a note without Mouse and survey them.

   Output: Survey results and screen recording that will provide insight to user habits.

   How test will be performed: We will have an external user create a note and then try to add multiple shapes, text and formatted text to the note. All of this should be done without the user using the mouse. We will record their actions and ask them some survey questions once they are done.

4.2.3 Area of Testing3 - Consistency of Open and Closed Notes.

1. test-NF212,215

   Type: Dynamic, Manual.

   Initial State: Application launched with an open empty note.

   Input/Condition: Adding shapes and text via keyboard short cuts

How test will be performed: See (test-F217, F243). We then compare it after opening to the one that was saved to see if there any differences. Additionally, we will attempt to close a note without saving and check it the autosave catches it

2. test-NF216

   Type: Manual, Static .

   How test will be performed: Code review on all parts that may use the network data, making sure each cant run in the background.

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.

Table 2: Traceability Matrix for Non-Functional Requirements

NF211	Low-latency input and rendering (less than *DISPLAY_LATENCY*. ms response).	Test-NF211: Verify that users can successfully make changes and have them appear within the allowed time.
NF218	Performance at scale (*SHAPE_LOAD_TEST_COUNT*.+ elements).	Test-NF211,Test-NF218: Verify that the program still functions and is responsive with large amounts of data in notes.
NF213	Cross-platform support (Windows/macOS/Linux).	Test-NF213: Verify program functionality on multiple different platforms.
NF214	Accessible keyboard shortcuts and discoverable command palette.	Test-NF214: Ensure users can find and use shortcuts provided by the program.
NF217	Visual consistency and intuitive UI layout.	Test-NF217: Run a usability survey with another user to check if the program is intuitive.
NF212	Local data persistence and autosave.	Test-NF212,215: Check that auto save create notes that are identical to he ones saved.
NF215	File integrity during save/export (atomic operations).	Test-NF212,215: Check that exports to different file formats properly display and regular saving.
NF216	Security of local data (no background network traffic).	Test-F216: Check that all code related to network traffic will not run in the background.

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

:

DISPLAY_LATENCY - 30. Expected Latency in milliseconds.

SHAPE_LOAD_TEST_COUNT - 500. Amount of shapes used for stress testing.:

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]

1. How difficult was it to find the short-cuts you needed?

2. What short-cuts felt the least intuitive?

3. What short-cuts felt the most intuitive?

4. Are their any missing short-cuts that you would want to have?

5. Are their any short-cuts that feel to burdensome or require too many inputs to execute?

6. Are any short-cuts that you accidentally pressed when you didn't want them?

7. How would you compare this to other note taking methods?

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 - I worked on the system tests section as well as the traceability matrix that connects the tests to the functional requirements. I think the team collaborated well to ensure that the tests covered all necessary aspects of the system. Also, I got more familiar with the SRS document while writing the tests, which helped me understand the project better. After the previous deliverables, me and the team both got more comfortable with the LaTeX format and GitHub workflow, which made the writing process smoother.
    - Ethan - I was responsible for writing out parts of the plan section involving the verification plans and test tooling. Something that went well while working on this was doing some more research and deciding the technologies we would use for testing. This gave everyone a better idea of the types of testing we would need to

do, the infrastructure that will need to be set up for them, and the frameworks we would use. It was also good to discuss this plan with the team and make sure to set (realistic) expectations for testing.

- Hussain - In this deliverable, I was tasked with writing the general information section of the VnV plan. To successfully write this section, not only did I have to understand the purpose of the VnV plan, but I had to review our previous documents such as the SRS, Problem Statement, and Hazard Analysis, to ensure consistency across our project's documentation. This allowed me to see how different documents interrelate and reinforced my understanding of the project's overall structure and objectives.
- Jeffrey - During this deliverable, I found that writing sections that referenced or related to the SRS and Hazard Analysis went very well. Since we put in extensive work into our SRS and Hazard Analysis, I found that working on the VnV (which builds on the SRS and other relating documents a bit) was very smooth. Since the SRS was well organized and most of the members were familiar with it's contents, we were able to efficiently reference it and complete the VnV Plan at a good pace.
- Kevin -What went well during this deliverable is our groups ability to communicate and assign work. We managed to have everyone assigned parts of the VnV document much in advance giving us time work on it. This also let us look over the VnV, estimating the amount of work expected, and properly plan on time to finish it.

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

    - Chengze - I initially had trouble aligning the LaTeX tables and ensuring proper formatting. I resolved this by researching the float and tabularx environments and asking for peer feedback. Additionally, I got confused on merge the pull requests when I need to make sure my branch is up to date with the main branch. I resolved this by learning how to rebase my branch onto the main branch before merging. Which is a result of smooth communication with team lead Ethan.

- Ethan - Honestly, this deliverable went pretty smoothly. There were some issues that arose with the GitHub Actions build system that were discovered while working on the last deliverable, but I took some time to fix them. I had felt that I was doing a lot of "intro" type work in the past deliverables so I wanted to switch to a different section. Hussain gladly agreed to take on the intro section for this doc and I did something different with the testing plan.
- Hussain - When looking at the VnV plan, I was initially confused on a few of the sections, particularly on what symbolic parameters were and how challenge levels worked. To resolve this, I asked about these things in our initial TA meeting for the deliverable. Symbolic parameters were explained to us as any constants that we would want to define in one place for easy maintenance and used throughout the document. Challenge level was a previously used concept in past years' VnV plans that was not relevant for this year's deliverable.
- Jeffrey - During this deliverable, I found that coordinating with the team was a little more difficult that usual. This is due to the fact that a lot of us had other course work and midterms happening during the creation of our VnV Plan. In order to resolve this issue, the team decided that each member should list their availability and schedules so we could properly organize a date and time that works best for everyone. In the past, it was a bit easier where we could propose a day and see if the team is available. This time around, we had to check availability first before proposing a day to meet. I found this solution to be very effective and will definetely use this when our schedules get busier.
- Kevin - The main pain point of this deliverable was balancing it along with all of the other work I have from other classes. I had to find time to finish this deliverable along with other assignments, midterms, labs and mettings for other projects.

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 - As the Quality Assurance Tester, I will focus on improving my ability to design and execute effective test cases that ensure all features meet the SRS specifications. First, I plan to strengthen my practical testing skills by using tools like Jest and

Playwright to create and run both manual and automated tests, which will help me better identify issues and verify functionality. Second, I aim to refine my understanding of QA methodologies by studying professional testing resources and documentation practices to improve how I record and communicate test results. Since my role centers on maintaining the reliability and quality of the product, these skills will help me ensure that our software performs as intended and meets all functional requirements.

- Ethan - I will be picking up the code coverage metric integration work. For acquiring this skill, two approaches would be reading the Coveralls documentation and watching Youtube tutorials. I will likely go the route of Youtube tutorials as I find them easier to follow. However, I will definitely use the documentation as well if I feel it will give me better/missing information.
- Hussain - Writing unit tests and component testing is the skill mentioned in the previous question that I will be focusing on. To acquire knowledge in this area, researching specific elements of UI components that need to be tested will be helpful, as our project is primarily a user interface and will most likely have standard components that require testing. Practicing writing unit tests different components is also a great way to learn this skill. By picking a real react project and using the outlined tools to write tests for its components, I can gain hands-on experience and understand how to effectively test the components of our own project. If time permits, I will attempt to approach learning this skill through practicing writing the unit tests, as I believe applying the knowledge in a practical manner will help me understand the concepts better than just researching them.
- Jeffrey - I would like to focus on becoming more familiar with CI/CD. Since most of us have not created or maintained one before, I feel like it's an important skill to develop, seeing as we cannot rely on any one member who already has expertise in this field. An approach that I plan on using to acquire the knowledge for this skill is to leverage my internship connections. During my co-op I worked closely with CI/CD team members in my department. While I never specifically worked on CI/CD, I was able to keep in touch with members from that time. For this capstone, I

plan on leveraging these connections to gain the skills that would benefit this capstone project. Additionally, I plan on improving my automation skills by developing a better understanding of how to build an automated test pipeline with GitHub actions. This is because I was assigned the role of Automation Tester when working on the Verification and Validation team. Since I was assigned this role, I feel inclined to learn this skill to contribute and fulfill my responsibilities as an Automation Tester. To acquire this skill, I've found many resources regarding how to build automated testing pipelines with GitHub actions. Since this isn't a niche skill or topic, there are many resources, such as guides or videos, that will assist me in acquiring this skill.

- Kevin - I would like to focus on building an automated testing pipeline with GitHub actions. As our project already had some issues with GitHub actions. Currently it attempts to compile the latex files. We had issues where we were doing bad merges and GitHub managed to catch them and display errors. By understanding how this works we can have a more robust system that can not only just test our documentation, but allow us to make sure that new features don't cause breaking changes. To acquire this skill, Place I could find information would be GitHub's official website which provides an explanation and documentation on how to implement actions onto our repository. If I have any errors or specific questions I can look on websites like stackOverflow that are more dedicated to solving errors. I will most likely use the documentation as my primary method of learning and implementing GitHub actions as it would contain the most general information on how to get started.