System Verification and Validation Plan for Software Engineering

Team 8, RLCatan
Rebecca Di Filippo
Jake Read
Matthew Cheung
Sunny Yao

October 27, 2025

Revision History

Date	Version	Notes
10/19/2025	Draft	Initial VnV plan without Unit Testing
...	...	...

Contents

List of Tables

List of Figures

1 Symbols, Abbreviations, and Acronyms

symbol	description
T	Test

The following glossary headers are hyperlinked to their online definitions for further reading.

- Catan – A strategy board game called *Settlers of Catan* where players collect resources, build roads/settlements, and trade to earn points.[1]
- AI – Field of computer science and engineering that focuses on creating systems capable of performing tasks that usually require human intelligence.[2]
- RL – A type of machine learning where an agent, known as Reinforcement Learning, learns by interacting with an environment and receiving rewards or penalties for its actions.[3]
- Digital Twin – A digital system that mirrors a physical one. In this project, it refers to a digital representation of the physical *Catan* board, updated in real time.[4]
- CV – An area of AI that trains computers to interpret and understand visual information, such as the physical *Catan* board.[5]
- LLM – A machine learning model trained on large amounts of text data to generate and understand language.[6]
- Game State – The current configuration of the game, including player resources, board layout, and dice rolls.[7]
- FR – A requirement that defines a function of the system; it specifies what the system must do.[8]
- NFR – A requirement that specifies a quality attribute of the system; it defines how the system must be.[9]

This document presents the Verification and Validation (V&V) plan for RLCatan. The plan will be updated and refined after the MIS (detailed design document) is completed.

2 General Information

2.1 Summary

The software being tested is RLCatan, an *AI* system that learns to play the board game *Catan* competitively through reinforcement learning. It includes a learning agent trained in a simulated environment, a computer vision module for detecting and interpreting the physical game board, and a interface that displays the game state and *AI*-generated move suggestions to users. The general purpose of RLCatan is to provide optimal move suggestions to human players based on real-time analysis of the game state.

2.2 Objectives

The objective is to build confidence in the correctness, reliability, and functional performance of the RLCatan system. Verification will focus on ensuring that the reinforcement learning agent generates valid moves, that the computer vision module correctly interprets game states, and that data exchange between system components is accurate and consistent.

Out of scope are extensive usability studies and large-scale performance benchmarking, as these exceed the project's time and resource constraints. Additionally, external dependencies such as Catanatron, OpenCV, and YOLOv9 are assumed to be verified by their developers, and will only be tested for correct integration within RLCatan.

2.3 Challenge Level and Extras

The challenge level for this project is Advanced, as outlined in the problem statement and confirmed with the course instructor. This project reflects the complexity of developing a reinforcement learning agent capable of mastering *Settlers of Catan*, integrating computer vision for real-time game state analysis, and ensuring seamless interaction between system components.

The approved extras for this project include:

- User Instructional Video: A walkthrough video that explains how to set up, run, and interact with the software, helping users understand its functionality and workflow.
- Performance Report: A comprehensive report summarizing the *AI*'s performance metrics, learning progression, and evaluation results based on various test scenarios.

2.4 Relevant Documentation

The following documents provide essential background for the VnV plan:

1. **Software Requirements Specification (DiFilippo et al. (2025)):** This document outlines the functional and non-functional requirements for RLCatan, serving as the primary reference for the verification and validation plan.

3 Plan

This section outlines the strategy for Verification and Validation (VnV) across all phases of the RLCatan project lifecycle. The plan details the roles and responsibilities of the VnV team, and specifies structured approaches for verifying the Software Requirements Specification (SRS), Design, and Implementation, as well as the strategy for Software Validation. This systematic approach is designed to increase confidence in the correctness, functional performance, and reliability of the final system. (insert visual roadmap)

3.1 Verification and Validation Team

The VnV efforts will be a collaborative team effort, with specific responsibilities assigned based on existing project roles to ensure systematic coverage and accountability.

Table 1: Verification and Validation Team Roles

Team Member	**Project Role**	**VnV Responsibilities**
Jake Read	Team Leader	Coordinating all VnV activities, performing final review of all test cases, and leading the code walkthrough for the supervisor.
Rebecca Di Filippo	Notetaker	Documenting VnV meeting minutes, maintaining traceability tables, and preparing the Usability Survey for validation.
Sunny Yao	IT/DevOps	Implementing and maintaining CI/CD pipelines, monitoring automated test coverage metrics, and troubleshooting technical VnV tool issues.
Matthew Cheung	Researcher	Researching and applying non-dynamic testing techniques (e.g., formal code inspection checklists) and collecting external data for software validation.
Dr. Istvan David	Project Supervisor	Providing technical oversight, expert consultation on *RL* model correctness, and participating in the Revision 0 demonstration for requirements validation.

3.2 SRS Verification

The SRS will be verified against the project's objectives to ensure all requirements are unambiguous, correct, feasible, and complete (Quality System Tests).

- **Structured Review with Supervisor:** A dedicated meeting will be scheduled with Dr. Istvan David, the Project Supervisor. The team will present the consolidated functional, non-functional, and safety requirements, specifically focusing on the formalizations (e.g., Z-Notation) and the key constraints (e.g., ≤ 5 second response time for advice). Specific questions will be prepared beforehand to validate the technical accuracy of the *RL* requirements (e.g., reward function structure).
- **Peer Review and Inspection:** Classmates and primary reviewers

will be engaged to perform an ad-hoc review of the SRS, targeting clarity and consistency.

- **Issue Tracker for Traceability:** All feedback and identified issues will be logged on the GitHub Issues board, assigned to a team member, and traced back to the specific requirement for resolution.
- **Checklist-Based Inspection:** The team will employ a custom checklist (developed by the Researcher) focusing on the characteristics of a high-quality requirement (unambiguous, verifiable, complete) to systematically inspect the SRS.

3.3 Design Verification

Design verification will occur after the Design Document is completed (Milestone: Nov. 10) to ensure the architectural design meets all requirements and is ready for implementation.

- **Interface Inspection:** The Researcher will lead an inspection of the component interfaces (e.g., the data contract between the Computer Vision Model and the Game State Manager) defined in the Design Document. This ensures modularity and that the design supports FR and NFR related to communication and data exchange.
- **Structured Peer Review:** The design will be reviewed by all team members to check for feasibility, proper component decomposition, and adherence to design principles (e.g., modularity).
- **External Review:** Classmates will be asked to review the design, focusing on potential bottlenecks or single points of failure, particularly regarding real-time operation and *AI* performance constraints.

3.4 Verification and Validation Plan Verification

The VnV Plan itself is a critical project artifact that must be verified for feasibility, clarity, and completeness.

- **Team Review:** The plan will be reviewed by all team members, ensuring that the defined VnV tasks are feasible within the academic timeframe and that the level of detail is sufficient for execution.

- **Feasibility Assessment:** The Team Leader will specifically check that the execution of the full VnV plan is realistic given the eight-month project timeline and the team's current skill set.
- **External Feedback:** The plan's clarity and completeness will be verified by the course TA and classmates during informal review sessions.

3.5 Implementation Verification

Implementation verification is focused on ensuring the source code correctly implements the design and fulfills the requirements. This will use a combination of dynamic (automated tests) and non-dynamic (static analysis and inspection) techniques.

- **Unit Testing and System Tests:** All functional and non-functional requirements will be verified through the comprehensive dynamic tests described in the System Tests and Unit Test Description sections of this document.
- **Static Analysis and Linters:** The team will enforce coding standards (PEP 8 for Python, Google Style Guide for JavaScript/React) using automated linters and static analyzers (SonarQube).
- **Code Walkthrough:** The final class presentation in CAS 741 (Final Demonstration, March 23) will serve as a high-level code walkthrough for a portion of the system, focusing on key components like the *CV* integration or *AI* decision-making process.
- **Peer Code Inspection:** All code changes will require a review and approval via GitHub Pull Requests. Team members will specifically check for adherence to coding standards, algorithm correctness, and proper error handling.

3.6 Automated Testing and Verification Tools

Automated VnV tools are central to maintaining code quality and continuous integration.

- **Unit Testing Frameworks:** Python's built-in `unittest` or `pytest` will be used for backend unit testing. Jest will be used for the frontend React components.

- **Continuous Integration (CI):** GitHub Actions will be implemented to run automated tests, linters, and build checks upon every push or pull request.
- **Static Analysis:** SonarQube will be used to analyze the codebase, identify code smells, potential bugs, and security vulnerabilities.
- **Code Coverage:** Tools like Coverage.py (for Python) will be integrated into the CI pipeline to report and track unit test coverage metrics, aiming for a minimum of 80% coverage for core logic modules.
- **Linters:** `flake8` will enforce the PEP 8 standard for all Python backend code. ESLint will enforce the Google Style Guide for the JavaScript/React frontend.

3.7 Software Validation

Software validation ensures the final product meets the actual needs of the end-users (players) and stakeholders.

- **Revision 0 Demonstration with Supervisor:** The Revision 0 Demonstration (Feb. 2) will be specifically used to validate that the core functionality and *AI* performance meet the project goals. Dr. Istvan David will provide critical feedback on the *AI*'s strategic viability and adherence to the project scope.
- **User-Centric Validation (Usability Survey):** A partial usability survey will be conducted (using the final class presentation or an informal user testing session with competitive *Catan* players) to validate the *NFR* related to usability (NFR.S.2) and the clarity of the *AI* advice. Survey questions will be included in the Appendix.
- **External Data Validation:** The *RL* agent's performance will be validated longitudinally by running thousands of game simulations and comparing its win rate and strategic output against existing benchmark *Catan* opponents (e.g., baseline bots), and potentially against human gameplay data provided by competitive players. This output will be summarized in the **Performance Report** extra.

4 System Tests

This section details the system-level tests for validating the functional and non-functional requirements of the RLCatan system. The system tests are designed to be run once the integrated system is complete, using real-world or representative data to simulate actual use cases. Traceability to specific requirements (e.g., $FR.S.1.1$) is maintained throughout.

4.1 Tests for Functional Requirements

This subsection outlines test cases organized by the primary functional component responsible for the requirement. These tests collectively cover all functional requirements detailed in Section S.2 of the SRS, including the core components: Computer Vision, *RL* Model, and Game State Management. (add more references maybe)

4.1.1 Area of Testing1

4.2 Tests for Functional Requirements

This subsection details the system tests for all functional requirements (*FR*) outlined in the SRS (Section S.2). The tests are categorized by the major functional areas of the RLCatan system to ensure comprehensive coverage. The Test IDs maintain clear traceability to their corresponding requirements.

4.2.1 AI Model and Recommendation Logic

This area focuses on verifying the core *AI* functionality, including strategy prediction ($FR.S.3.1$), logical evaluation of moves ($FR.S.3.2$), and integration with the *RL* Environment ($FR.S.2.5$) and the official game rules ($FR.S.2.1$). These tests ensure the *AI* provides optimal, valid, and context-aware suggestions.

Test Case: *AI* Move Validity and Strategy Prediction

1. T.FR.AI.1.1
   Control: Automatic

   Initial State: A game state is loaded where Player A has exactly 1 Wood, 1 Brick, and 2 Roads left to build. All adjacent spaces for a new road are occupied by other players' structures, leaving only one legal

road placement location. Player A has no other legal moves (cannot build a settlement, city, or development card).

Input: The structured *GameState* is sent to the *AI* model via the Game State Manager ($FR.S.2.5$).

Output: The *AI*'s move suggestion must be `BuildRoad` at the single remaining legal location. No illegal moves (e.g., `BuildSettlement` or suggestions to trade without partners) should be returned.

Test Case Derivation: Verifies $FR.S.3.1$ (Strategy Prediction) and $FR.S.3.2$ (Evaluate potential moves) by enforcing a state with only one legal optimal move, thus checking adherence to rules ($FR.S.2.1$) and logical constraints.

How test will be performed: An automated simulation within the *RL* Environment ($FR.S.2.1$) is run 10 times from this exact pre-set state. All 10 runs must result in the suggestion of the single legal road placement.

2. T.FR.AI.1.2
Control: Manual

Initial State: *GameState* where Player A is 1 Victory Point (VP) away from winning (e.g., 9 VP + Longest Road) and has enough resources to build a settlement, but not a city. An open, unblocked node that grants the winning 10th VP is available.

Input: *GameState* where the immediate winning move is available.

Output: The *AI* must provide the suggestion to `BuildSettlement` at the winning node, with a high confidence score ($FR.S.3.4$). The textual reasoning should explicitly state that this move secures the victory.

Test Case Derivation: Verifies $FR.S.3.1$ (Strategy Prediction) by testing the *AI*'s ability to identify and prioritize an immediate game-winning condition over resource maximization.

How test will be performed: Load the state and manually check the UI display ($FR.S.4.4$) to ensure the winning move is highlighted and prioritized over all other options.

3. T.FR.AI.1.3
Control: Automatic

Initial State: *GameState* loaded for a 4-player game where the current player (Player A) has 6 resources and rolls a 7, triggering the Robber phase.

Input: The structured *GameState* is fed to the *AI* model.

Output: The *AI* must correctly calculate that Player A needs to discard 3 cards (half of their resources, rounded down), and must suggest a move (placing the Robber, stealing a card) that is fully legal, verifying the rule simulation ($FR.S.2.1$).

Test Case Derivation: Checks a critical game transition rule (Robber movement/discarding) against the simulated environment and *AI* logic, ensuring adherence to the core ruleset.

How test will be performed: Run 10 simulations where the Robber is rolled. Verify the system logic correctly applies the discarding rule and the *AI* suggests a valid Robber placement move.

4.2.2 User Interface and Data Management

This area verifies that the system correctly manages, displays, and archives game data, supporting both real-time user interaction ($FR.S.4.1$ to $FR.S.4.6$) and persistent data storage ($FR.S.5.1$ to $FR.S.5.5$). This ensures data integrity and user experience.

Test Case: Real-time UI Update and Database Integrity

1. T.FR.UI.2.1
   Control: Manual/Automatic

   Initial State: Game has just started. Player A has 2 settlements and 2 roads. The UI displays Player A's resource inventory as empty.

   Input: Player A performs the action `BuildRoad` at node X, which is processed by the Game State Manager ($FR.S.7.4$) and updated in the *GameState* Database ($FR.S.5.1$) using the Image Queue ($FR.S.6.1$).

   Output: a) The UI must immediately reflect the change, showing the new road on the board visualization ($FR.S.4.1$) and Player A's resource inventory accurately reduced ($FR.S.4.2$). b) The *GameState* Database entry for this turn must be complete, correctly logging the pre-move state, the move performed, and the resulting post-move state ($FR.S.5.2$).

Test Case Derivation: Verifies the real-time data flow requirements ($FR.S.6.1$) and the system's ability to automatically and accurately update all components after a player action ($FR.S.7.4$).

How test will be performed: Perform 5 consecutive build actions and measure the time for the UI to update. After the sequence, query the *GameState* Database to confirm that 5 distinct, consistent, and sequentially correct game states were logged ($FR.S.5.1$).

2. T.FR.UI.2.2
Control: Automatic

Initial State: Game is 50 turns long and complete. The game state database contains the full history.

Input: Request historical game data ($FR.S.5.5$) for the completed game, specifically requesting all dice rolls and all development card purchases.

Output: The system successfully returns the complete list of 50 turn records, including the dice roll for each turn, and a timestamped record of every development card purchased, demonstrating access to historical data ($FR.S.7.3$).

Test Case Derivation: Verifies the requirements for historical logging and structured access to past game data ($FR.S.5.5$).

How test will be performed: An automated script runs 10 full simulated games, each over 40 turns. After each game, the script attempts to query the database for the complete turn log. The test passes only if all 10 logs are complete, consistent, and accurately reflect the game rules.

4.2.3 Computer Vision and Game State Capture

This area focuses on verifying the accuracy and reliability of the Computer Vision (CV) Model (*FR*.S.1.x) and the Game State Manager's (*FR*.S.7.x) ability to convert visual input into a consistent digital representation (*GameState*). The tests ensure the system can accurately detect board elements ($FR.S.1.1$), translate them into structured data ($FR.S.1.2$), and maintain synchronization ($FR.S.7.1$).

Test Case: Initial Board Setup Recognition

1. T.FR.CV.3.1
   Control: Manual

   Initial State: A standard, legal *Catan* board is set up for the initial placement phase. Resource tiles, number tokens, and the Robber are placed according to the rules. No settlements or roads have been placed.

   Input: A captured image (or video frame) of the physical board.

   Output: A structured *GameState* data object where the board topology (tile types and locations, number tokens, Robber position) is correctly mapped to the digital twin. The asset counts for all players (settlements: 5, roads: 15) must be correct, verifying $FR.S.7.2$.

   Test Case Derivation: Directly verifies $FR.S.1.2$ (Feature-to-State Translation) and $FR.S.7.1$ (State Synchronization) using a known baseline state.

   How test will be performed: A set of 5 different, legally generated board setups will be used. For each setup, the system is fed the image, and the output *GameState* dictionary is inspected to ensure a 100% match with the actual setup.

2. T.FR.CV.3.2
   Control: Manual

   Initial State: A game is mid-play. Player A has 2 settlements, 1 city, 8 roads. The camera is slightly misaligned, causing a minor occlusion of one edge.

   Input: A captured image of the board with the occlusion.

   Output: The system successfully detects the presence of Player A's 2 settlements, 1 city, and 8 roads, correctly applying logical constraints to infer the state despite the occlusion. The system provides a diagnostic confidence metric ($FR.S.1.5$) for the occluded area.

   Test Case Derivation: Verifies $FR.S.1.3$ (Error Detection and Correction) and $FR.S.1.5$ (Diagnostic Feedback) under sub-optimal real-world conditions.

   How test will be performed: Artificially introduce a small, known occlusion (e.g., a hand partially covering an intersection) in a mid-game state. Check that the digital state is correctly reconstructed and

that the confidence metric correctly flags the occluded element as low-confidence/corrected.

4.3 Traceability Between Test Cases and Requirements

The table below documents the explicit traceability between the defined system test cases ($T.ID$) and the specific requirements ($Req.ID$) they verify. This ensures that every functional and critical non-functional requirement is covered with appropriate redundancy, building confidence in the product if all tests pass.

Table 2: Functional Requirement Test Case Traceability Matrix

Test Case ID	**Requirements Verified (Subset)**
T.FR.AI.1.1	$FR.S.3.1$, $FR.S.3.2$, $FR.S.2.1$, $FR.S.2.5$, $FR.Sa.3$
T.FR.AI.1.2	$FR.S.3.1$, $FR.S.3.4$
T.FR.AI.1.3	$FR.S.2.1$
T.FR.UI.2.1	$FR.S.4.1$, $FR.S.4.2$, $FR.S.5.1$, $FR.S.5.2$, $FR.S.6.1$, $FR.S.7.4$, $FR.S.7.2$
T.FR.UI.2.2	$FR.S.5.5$, $FR.S.7.3$
T.FR.CV.3.1	$FR.S.1.2$, $FR.S.7.1$, $FR.S.7.2$
T.FR.CV.3.2	$FR.S.1.3$, $FR.S.1.5$

5 Tests for Nonfunctional Requirements

This section outlines the system-level tests for validating the nonfunctional requirements (NFRs) defined in Section S.2.8 of the SRS. These include scalability, usability, maintainability, installability, data integrity, and availability.

Each subsection corresponds to a major nonfunctional quality attribute of the RLCatan system. Some tests produce summary statistics or qualitative evaluations (e.g., graphs or surveys) rather than strict pass/fail outcomes.

Accuracy-related nonfunctional qualities are validated implicitly through the functional tests defined in Section 4.2 (T.FR.AI.2.1–2.3 and T.FR.CV.1.1–1.2), which record relative error and correctness metrics.

5.1 Performance and Scalability

This area ensures the RLCatan system maintains acceptable responsiveness and throughput when supporting multiple concurrent games.

Test Case: Scalability and Response Time Evaluation

1. T.NFR.P.1
   Control: Automatic

   Initial State: The RLCatan server is running the complete software stack (*AI*, *CV*, Database, UI).

   Input/Condition: Simulate 1, 3, and 5 concurrent games using the *RL* environment and mock UI clients.

   Output/Result: Average response time per game, CPU load, and memory usage are measured and summarized in a graph.

   Test Case Derivation: Verifies NFR.S.1 (Scalability) by demonstrating that response time increases sub-linearly with load.

   How test will be performed: Automated load-testing scripts measure latency between GameState updates and *AI* recommendations. Results are summarized in a table of performance metrics versus number of sessions.

5.2 Usability

This area focuses on confirming that the interface is intuitive and accessible to users with minimal training. It also includes accessibility checks for users with visual impairments.

Test Case: Usability Survey and User Feedback

1. T.NFR.U.1
   Control: Manual

   Initial State: Final prototype deployed in a browser.

   Input/Condition: 5–10 participants familiar with *Catan* complete a guided session and fill out the usability survey (Appendix 7.2).

   Output/Result: Average rating $\geq 4/5$ for clarity, ease of navigation, and understanding of *AI* feedback.

Test Case Derivation: Verifies NFR.S.2 (Usability) through direct user evaluation.

How test will be performed: Conducted during the final demo; results aggregated and visualized as a bar chart.

Test Case: Accessibility and Visual Clarity

1. T.NFR.U.2
   Control: Manual

   Initial State: RLCatan UI loaded in browser.

   Input/Condition: Apply color-blindness filters and vary display scaling.

   Output/Result: All critical UI elements remain legible; contrast ratio $\geq 4.5 : 1$.

   Test Case Derivation: Verifies accessibility and clarity under varied visual conditions.

   How test will be performed: Tested using Chrome DevTools accessibility simulator; screenshots recorded for documentation.

5.3 Maintainability

This area ensures the modular design and structure of the codebase support efficient updates and debugging.

Test Case: Code Quality and Modular Design Inspection

1. T.NFR.M.1
   Control: Static

   Initial State: Complete source code committed to GitHub.

   Input/Condition: Run `SonarQube` and `flake8` to generate maintainability and style reports.

   Output/Result: No high-severity issues; modules follow consistent structure and clear separation of concerns.

   Test Case Derivation: Verifies NFR.S.3 (Maintainability) through code inspection and static analysis.

   How test will be performed: Peer inspection led by group members; issues logged in GitHub for resolution.

5.4 Installability

This area validates that RLCatan installs and operates correctly across all supported operating systems and browsers.

Test Case: Cross-Platform Installation Validation

1. T.NFR.I.1
   Control: Manual

   Initial State: Packaged release available.

   Input/Condition: Install and execute RLCatan on Windows 11 and macOS using Chrome and Firefox browsers.

   Output/Result: Application installs and runs successfully on all tested platforms.

   Test Case Derivation: Verifies NFR.S.4 (Installability) by ensuring cross-platform compatibility.

   How test will be performed: Each team member installs on a unique OS following a checklist; issues recorded and documented.

5.5 Data Integrity

This area ensures that the Game State Database maintains consistent, valid, and accurate data under all conditions, including normal operation, concurrent updates, and unexpected failures. This includes validation of incoming updates and transactional consistency to prevent corruption.

Test Case: Transaction Integrity and Recovery

1. T.NFR.D.1
   Control: Automatic

   Initial State: Active game session connected to the database.

   Input/Condition: Execute 10 consecutive state updates with an artificial network interruption during one update.

   Output/Result: Database remains consistent; interrupted transaction is rolled back cleanly; no other game states are affected.

   Test Case Derivation: Verifies transactional consistency portion of NFR.S.5 under failure conditions.

How test will be performed: Automated integration test simulates network failures and verifies database consistency after each run.

Test Case: Data Validation on Updates

1. T.NFR.D.2
   Control: Automatic

   Initial State: Active game session connected to the database.

   Input/Condition: Attempt to submit invalid game state updates, such as negative resource counts, illegal moves, or missing required fields.

   Output/Result: Invalid updates are recognized and rejected.

   Test Case Derivation: Verifies validation portion of NFR.S.5, preventing data corruption from malformed inputs.

   How test will be performed: Automated tests inject invalid game state objects and verifies rejection.

Test Case: Concurrent Update Handling

1. T.NFR.D.3
   Control: Automatic

   Initial State: Two or more simultaneous game sessions active.

   Input/Condition: Simulate concurrent updates to overlapping resources (e.g., two players updating the same tile).

   Output/Result: Only one valid transaction is committed; conflicts are detected and resolved; no data corruption occurs.

   Test Case Derivation: Verifies database integrity under concurrent operations, completing NFR.S.5 coverage.

   How test will be performed: Automated integration tests run multiple simulated clients updating shared resources; system logs and database state are verified for correctness.

5.6 Availability

This area verifies that the system automatically recovers from failures within one minute, as required in the SRS.

Test Case: System Failure and Auto-Recovery

1. T.NFR.AV.1
   Control: Manual

   Initial State: System running with active game session.

   Input/Condition: Force a server crash or database disconnect mid-turn.

   Output/Result: System restarts and restores previous game state within 60 seconds.

   Test Case Derivation: Verifies NFR.S.6 (Availability) by measuring recovery time following induced failure.

   How test will be performed: Use shell script to crash the process, record downtime using system logs, verify recovery from last snapshot.

5.7 Accuracy and Correctness (Referenced)

Accuracy-related NFRs are verified by functional tests defined in Section 4.2:

- T.FR.AI.2.1–2.3 (*AI* Model and Recommendation Logic)
- T.FR.CV.1.1–1.2 (Computer Vision and Game State Capture)

These tests report relative error between predicted and expected outcomes. No separate nonfunctional test cases are required.

5.8 Traceability Between Test Cases and Requirements

Table 3: Non-Functional Requirement Test Case Traceability Matrix

Test Case ID	**Requirement(s) Verified**
T.NFR.P.1	NFR.S.1 – Scalability
T.NFR.U.1	NFR.S.2 – Usability
T.NFR.U.2	NFR.S.2 – Accessibility
T.NFR.M.1	NFR.S.3 – Maintainability
T.NFR.I.1	NFR.S.4 – Installability
T.NFR.D.1	NFR.S.5 – Data Integrity
T.NFR.D.2	NFR.S.5 – Data Integrity
T.NFR.D.3	NFR.S.5 – Data Integrity
T.NFR.AV.1	NFR.S.6 – Availability

6 Unit Test Description

This section is to be completed after the MIS (detailed design document) has been completed.

References

Rebecca DiFilippo, Jake Read, Matthew Cheung, and Sunny Yao. System requirements specification. https://github.com/SY3141/RLCatan, 2025.

7 Appendix

This is where you can place additional information.

7.1 Symbolic Parameters

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

Table 4: Symbolic Constants for System Tests

Constant	Value	Description
T_RESP	5 seconds	Maximum allowed time for the *AI* to display a move recommendation (*NFR.E.*4, *NFR.Sa.*1).
N_BATCH	100	Number of *GameState* objects used in a batch test for latency measurement.
N_RECONNECT	3	Minimum number of automatic reconnection attempts after connection loss (*FR.Sa.*6).
N_BOARD_SETUP	5	Number of legally generated board setups used to test CV initialization.
N_SIM_RUNS	10	Number of simulation runs for verifying *RL* move validation and rule application.

7.2 Usability Survey Questions?

This section provides the planned questions for the usability survey, which will be used to validate the usability Non-Functional Requirements (*NFR.S.*2) and gather feedback on the clarity of the *AI* suggestions (*FR.Sa.*7), primarily targeting the Novice and Intermediate player profiles.

1. **Clarity of *AI* Suggestion:** When the *AI* suggested a move, how clear was the visual highlighting or overlay that indicated the location and action? (1=Very Unclear, 5=Very Clear)

2. **Ease of Interface Use:** How easy was it to navigate the RLCatan interface to access the current *GameState* and receive the *AI* advice? (1=Very Difficult, 5=Very Easy)

3. **Speed of Advice:** Did the advice appear fast enough to be useful during a normal turn of play? (Yes/No, If No: please explain.)

4. **Trust in System:** How much confidence do you have in the system's ability to accurately reflect the physical board state? (1=Very Low, 5=Very High)

5. **Usefulness of Post-Game Analysis:** If you used the post-game analysis, how helpful were the insights for improving your future strategy? (1=Not Helpful, 5=Extremely Helpful)

Appendix — Reflection

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?

Jake Read

1. This deliverable went fairly smoothly, mostly because we already had a clear set of requirements from the SRS to base the tests on. Unlike the previous deliverables, we split into two subteams. Matthew and

Rebecca worked primarily on this doc, while Sunny and I focused on getting started with the PoC, and then checked in with the VnV team to help review. I feel like this made the process more efficient, and less people working directly on one document meant fewer consistency issues.

2. Personally, I didn't have any major pain points with the VnV plan, but as I said before, I was mainly focused on the PoC and simply reviewed and helped organize VnV progress. I think Rebecca and Matthew did a great job, so the review process was quite smooth. It's possible they had some pain points I'm not aware of, but from my discussions with them this doc was more straightforward than the SRS.

3. For VnV, I think the team will need to acquire more knowledge about automated testing frameworks, particularly for Python and JavaScript. I'm not sure how experienced the others are in these areas, but when it comes to testing almost all my experience is in Java, so I think this will be a learning opportunity for me. We'll also need to familiarize ourselves with static analysis tools like SonarQube, as well as CI/CD pipelines using GitHub Actions. I can focus on SonarQube, since I'm interested in using it for future projects.

4. For learning about automated testing frameworks, we could either take online courses/tutorials or read official documentation and implement small projects to practice. I also previously used SonarQube briefly back in second year, so I could revisit that experience and supplement it with online resources. Of these options, I'll start with some online tutorials, and then reference what I did with some of my old work.

Rebecca DiFilippo

1. Writing this deliverable went fairly smoothly, mainly because we had a solid set of requirements from the SRS to guide the tests. Splitting into two subteams worked really well this time. Matthew and I focused on drafting the V&V document, while Sunny and Jake worked on the PoC and then helped review our work. I think this setup made things more organized and helped avoid overlapping edits and consistency issues. It also allowed us to make progress on the POC plan.

2. I didn't run into major pain points, though coordinating between the two subteams sometimes required a bit of back-and-forth to make sure everyone was aligned. Overall, Matthew and I were able to keep the doc consistent, and Jake and Sunny's feedback was really helpful for catching small issues early. The process felt much smoother than working on the SRS.

3. For V&V, the team will need to build more experience with automated testing frameworks.We also we'll need to get comfortable with static analysis tools like SonarQube and CI/CD pipelines through GitHub Actions. I'll focus on automated testing in Python since that's an area I want to strengthen for this project and future work which includes flake8 style checks.

4. To build these skills, we can look for online tutorials and courses, as well as consult with other people who have experience in these areas. I'll start with online tutorials for Python testing frameworks and flake8, as well as look for example projects that implement these tools effectively.