System Verification and Validation Plan for Re-ProtGNN

Yuanqi Xue

April 18, 2025

Revision History

Date	Version	Notes
Feb 24	1.0	Initial Draft

Contents

List of Tables

1 Symbols, Abbreviations, and Acronyms

symbol	description
T	Test
A	Assumption
R	Requirement
SRS	Software Requirements Specification
ProtGNN	Prototype-based Graph Neural Network
Re-ProtGNN	Re-implementation of the ProtGNN model
GNN	Graph Neural Network
GIN	Graph Isomorphism Network

This document outlines the Verification and Validation (VnV) plan for Re-ProtGNN, a prototype-based interpretable Graph Neural Network. The objective of this plan is to ensure that the system meets the functional and non-functional requirements specified in the Software Requirements Specification (SRS). The document is structured as follows: Section 2 provides general information about Re-ProtGNN, including its objectives and scope. Section 3 details the verification strategies, covering SRS verification, design verification, and implementation verification. Section 4 describes the system tests, including functional and non-functional testing. Finally, Section 5 will cover additional test descriptions as needed.

2 General Information

2.1 Summary

The software under validation is Re-ProtGNN, a model designed to enhance the interpretability of Graph Neural Networks. It aims to classify graph-structured data while generating prototypes that explain its predictions. The system consists of two main components:

- Training Phase: Learns meaningful representations of graphs while optimizing a loss function to improve both classification accuracy and prototype relevance.
- Inference Phase: Uses the trained model to classify unseen graphs and generate a set of representative prototypes that provide human-interpretable explanations.

Re-ProtGNN is implemented in Python, utilizing PyTorch Geometric for graph learning and Pytest for automated testing.

2.2 Objectives

The main goal of this VnV plan is to verify the correctness of the program, which includes ensuring that the system correctly processes graph-structured inputs, trains models effectively, and generates prototype-based explanations.

Out-of-Scope Objectives:

- External Library Verification: Core dependencies such as PyTorch and Torch-Geometric are presumed to be reliable and are not explicitly tested in this plan.
- Graph Encoder Validation: Graph encoders, including Graph Isomorphism Networks (GINs), are adopted from prior research, and their correctness is assumed without additional validation.

2.3 Challenge Level and Extras

This is a research project, and no additional components will be included due to time constraints.

2.4 Relevant Documentation

The Re-ProtGNN project is supported by several key documents that ensure the system is properly designed, implemented, and validated. These documents include:

- Problem Statement: This document (Xue, 2025g) introduces the motivation of Re-ProtGNN and the core problem it aims to solve.
- Software Requirements Specification (SRS): The SRS (Xue, 2024) defines the functional and non-functional requirements of Re-ProtGNN, and it also outlines the expected behavior of the system.
- Verification and Validation (VnV) Plan: The VnV Plan (Xue, 2025e) describes the testing strategy used to verify the system's functionality and evaluate its correctness.
- Module Guide (MG): The MG (Xue, 2025a) outlines the design decisions, responsibilities, and internal details of each module to support understanding and modular testing.
- Module Interface Specification (MIS): The MIS (Xue, 2025c) specifies the interface, access routines, and expected behavior of each module, enabling consistent implementation and verification.

3 Plan

This section outlines the Verification and Validation (VnV) plan for Re-ProtGNN. It starts with an introduction to the verification and validation team (Subsection 3.1), detailing the members and their roles. Next, it covers the SRS verification plan (Subsection 3.2), followed by the design verification plan (Subsection 3.3). The document then presents the VnV verification plan (Subsection 3.4) and the implementation verification plan (Subsection 3.5). Finally, it includes automated testing and verification tools (Subsection 3.6) and concludes with the software validation plan (Subsection 3.7).

3.1 Verification and Validation Team

Name	Document	Role	Description
Yuanqi Xue	All	Author	Create and manage all required documents, develop the VnV plan, conduct VnV testing, and verify the implementation.
Dr. Spencer Smith	All	Instructor/ Reviewer	Review all the documents.
Yinying Huo	All	Domain Expert	Review all the documents.

Table 1: Verification and Validation Team

3.2 SRS Verification Plan

An initial review of the SRS will be conducted by Dr. Spencer Smith and Yinying Huo to ensure its accuracy, completeness, and feasibility. The review will follow a manual inspection process using an SRS Checklist (Xue, 2025d) to assess the clarity and alignment of SRS with project objectives.

Reviewers will provide feedback via GitHub issues, and Yuanqi Xue, as the author, will be responsible for addressing revisions.

3.3 Design Verification Plan

The design verification process, covering the Module Guide (MG) and Module Interface Specification (MIS), will be conducted by Dr. Spencer Smith and domain expert Yinying Huo. Their feedback will be shared through GitHub issues, and Yuanqi Xue will be responsible for making the necessary revisions.

To ensure a structured verification process, Dr. Spencer Smith has created an MG checklist (Xue, 2025b) and MIS checklist Xue (2025b), both of which will be used to evaluate clarity, consistency, and correctness in the design documents. The verification process will ensure that the system architecture, module interactions, and design choices align with the project objectives.

3.4 Verification and Validation Plan Verification Plan

The Verification and Validation (VnV) Plan will be reviewed by Dr. Spencer Smith and domain expert Yinying Huo to ensure that the validation methodology aligns with the project's objectives. Feedback from reviewers will be provided via GitHub issues, where all necessary revisions and updates will be documented.

To maintain consistency and thorough evaluation, a VnV checklist (Xue, 2025f) prepared by Dr. Spencer Smith will be used to assess the completeness, correctness, and applicability of the verification and validation process.

3.5 Implementation Verification Plan

The implementation of Re-ProtGNN will be verified through a combination of unit tests and system tests. This verification process ensures that both functional and non-functional requirements are satisfied, as detailed in Section 4. Unit tests will target core components of the system, such as data preprocessing, model inference, and prototype generation, to confirm correctness at the module level. System tests will evaluate the end-to-end performance of Re-ProtGNN by validating input data, confirming training convergence, and verifying that the correct number of prototypes is generated. The specific testing tools and methodologies employed are described in Section 3.6, and the corresponding test cases are presented in Section 4.

3.6 Automated Testing and Verification Tools

Re-ProtGNN will use a combination of automated testing and verification tools to ensure code correctness:

Unit Testing: Pytest will be used for testing individual components, including data processing, loss functions, and inference logic. These tests will help verify that each module functions as expected before integration into the full system.

Continuous Integration (CI): GitHub Actions will be configured to automate testing after each commit. The workflow will include:

- Running unit tests to verify functionality.
- Performing data integrity checks.
- Validating dependencies to prevent compatibility issues.

3.7 Software Validation Plan

A testing dataset (i.e., a separate 20% test split from the MUTAG dataset (Debnath et al., 1991)) will be used to assess the model's classification effectiveness and the clarity of its prototype-based explanations, with accuracy serving as the main benchmark.

4 System Tests

This section outlines the system tests designed to evaluate both functional and non-functional requirements.

4.1 Tests for Functional Requirements

This section outlines the tests designed to validate the functional requirements of Re-ProtGNN, ensuring the system behaves as expected under various conditions. The testing areas include input verification, model training correctness, and inference validation, which corresponds to the R1, R2, and R3 of SRS (Xue, 2024).

4.1.1 Area of Testing 1: Input Validation

This section ensures the system can correctly detect and handle invalid or missing input files before dataset processing, as required to fulfill input robustness and error traceability.

Test for Raw Input Integrity

1. T1: Missing File Handling Test

   Control: Automatic

   Initial State: Dataset folder exists, but no raw input files are present.

   Input: A folder named MUTAG/raw with no content.

   Output: The system should raise a FileNotFoundError indicating which required raw file is missing.

   Test Case Derivation: Ensures that the system validates file existence prior to processing. This corresponds to preconditions for requirement R1.

   How test will be performed: A Pytest script sys_test_input_validation.py creates an empty raw input directory and initializes the MUTAG dataset wrapper. It passes if FileNotFoundError is raised and the error message references the correct file path.

2. T2: Format Consistency and Type Validation Test

   Control: Automatic

   Initial State: All required files are present but contain incorrectly formatted (non-numeric or corrupted) content.

   Input: A folder named MUTAG/raw with files such as MUTAG_A.txt and MUTAG_graph_labels.txt containing malformed strings.

   Output: The system should raise a ValueError if the file content is not parseable (e.g., containing strings instead of numerical values).

   Test Case Derivation: Ensures that the dataset loader enforces structural correctness, file consistency, and semantic type assumptions.

How test will be performed: A Pytest script sys_test_input_validation.py populates all required files with junk data (e.g., strings instead of integers) and verifies that a ValueError is raised during parsing or validation.

3. T3: Valid Dataset Load Test

   Control: Automatic

   Initial State: The full MUTAG dataset is properly placed under the ./data/MUTAG directory with correctly formatted raw files.

   Input: All required raw text files (adjacency, node labels, graph labels, and graph indicators) with valid data.

   Output: The dataset should load successfully, returning a valid dataset instance with length greater than zero. No exceptions should be raised.

   Test Case Derivation: Confirms the system can successfully load and parse a standard benchmark dataset, satisfying R1 under nominal conditions.

   How test will be performed: A Pytest script sys_test_input_validation.py initializes the dataset wrapper using the default data path and asserts the dataset loads correctly and contains nonzero graphs.

4.1.2 Area of Testing 2: Training Convergence

This test verifies that the model training process optimizes the objective function over time. It ensures that the model effectively minimizes training loss, thereby satisfying the requirement for stable and meaningful training behavior.

Test for Training Loss Convergence

1. T2: Loss Convergence Test

   Control: Automatic

   Initial State: The training pipeline is configured using the default parameters, and the training dataset is loaded from the dataset folder. No manual intervention is required.

Input: A graph classification dataset split into training, validation, and test sets; an untrained model initialized with random weights; and the configuration parameters defined in the system (e.g., maximum number of epochs, early stopping patience).

Output: A sequence of training loss values recorded after each epoch. The final loss should be less than or equal to the initial loss, indicating convergence.

Test Case Derivation: This test confirms that the model satisfies the training requirement by reducing the loss function across epochs, as expected in successful optimization procedures. If the training loss increases over time or fluctuates abnormally, the test will fail, revealing instability in training.

How the test will be performed:

- In the Pytest script sys_test_loss_converge.py, The system's main training pipeline is launched using the main entry point with loss coefficient values set to zero to isolate base training behavior.
- The append_record function is temporarily patched to intercept and store loss values across epochs in a list.
- Once training is complete, the list of recorded losses is analyzed.
- The test asserts the condition that the final loss must be less than or equal to the initial loss, indicating that training successfully reduced the loss.

A successful result can demonstrates that the system is capable of learning meaningful representations in the training phase.

4.1.3 Area of Testing 3: Inference Verification

This test ensures that the trained model can correctly classify unseen test samples and that the predictions returned during the inference phase are complete, numerically valid, and structurally consistent.

Test for Inference Accuracy

1. T3: Inference Accuracy Test

Control: Automatic

Initial State: A trained model checkpoint is saved in the src/checkpoint directory and available for loading.

Input: A test data loader produced from the configured dataset split, and a graph neural network model initialized and updated using the saved checkpoint weights.

Output: The system should return a scalar classification accuracy value, a scalar loss value, a 1D tensor of predicted class labels, and a 2D tensor or array of class probability scores for each test instance.

Test Case Derivation: This test verifies that the trained model satisfies R3 in SRS (Xue, 2024) by making predictions on the test set and reporting classification performance. It also confirms that the returned predictions and probabilities are well-formed and aligned with the dataset.

How test will be performed:

- In the Pytest script sys_test_input_validation.py, the test begins by loading the full dataset and applying a fixed ratio-based split into training, validation, and test sets using load_dataset().
- The model is constructed using the input and output dimensions inferred from the dataset and updated with the best checkpoint saved during training.
- Inference is performed on the test split using run_inference().
  - The returned dictionary contains the keys acc (accuracy) and loss, both of which must be real-valued scalars.
  - The accuracy must be strictly greater than 50%, which confirms that the model is performing above chance level on a binary classification task.
  - The shape of the prediction tensor all_preds must match the number of samples in the test set, meaning that there is exactly one predicted label for every input graph.
  - The probability tensor all_probs must be two-dimensional, with shape (N, C), where N is the number of test samples and C is the number of classes. This ensures that the model outputs a complete probability distribution for each test instance.

– The first dimension (number of rows) of all_probs must exactly equal the length of all_preds, confirming that the predicted label and associated probability scores correspond to the same set of inputs.

The classification accuracy is computed using the formula:å

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Data Points}} = \frac{1}{N}\sum_{i=1}^{N} 1(\hat{y}_i = y_i)$$

where N is the total number of graphs in the test set, $\hat{y}_i$ is the predicted label for the ith graph, and y_i is the corresponding ground-truth label. The indicator function $1(\cdot)$ evaluates to 1 if the predicted label is correct, and 0 otherwise.

4.2 Tests for Nonfunctional Requirements

This section outlines the tests designed to verify the nonfunctional requirements of Re-ProtGNN, including accuracy, usability, and portability.

4.2.1 Reliability

The reliability of the software is demonstrated through the execution of functional requirement tests, as described in Section 4.1 and Section 5.2. These tests ensure the system behaves as expected under defined conditions.

4.2.2 Usability

The usability of the software is measured mainly by the Usability Survey in Section 6.1.

1. T4: Usability Survey Test
   Type: Manual
   Initial State: The software is fully set up and operational
   Input/Condition: None
   Output/Result: Completed survey responses from the user
   Test Case Derivation: Measures user perception of ease of use and clarity of the interface, as defined by usability metrics.
   How test will be performed: The user will be asked to complete a survey after using the system. The survey questions are listed in Appendix 6.1.

4.3 Traceability Between Test Cases and Requirements

	T1	T2	T3	T4
R1	X			
R2		X		
R3			X	
NFR1	X	X	X	
NFR2				X

Table 2: Traceability Matrix of Test Cases and Requirements

5 Unit Test Description

5.1 Unit Testing Scope

This section outlines the unit testing coverage across the module hierarchy presented in Section 5. Unit tests were implemented for all leaf modules that contain self-contained functionality. Specifically, we provide full coverage for:

- **M3 - Input Format Module:** This module includes dataset loading, preprocessing, and dataloader construction. Unit tests validate both correct data splits and failure cases such as missing or malformed input files.
- **M5 - Training Module:** This module governs the model training procedure, including loss computation, prototype projection, mode configuration, and evaluation. Unit tests exercise all internal access routines except for the top-level train() function, which is validated through full-system tests (e.g., loss convergence test in 4.1.2).
- **M6 - Output Visualization Module:** This module handles logging, checkpoint saving, and subgraph visualization. Unit tests verify file-writing routines, model serialization behavior, and visual explanation dispatching logic for supported datasets.

- **M8 - Inference Module:** This module runs model evaluation on test data and reports metrics. Unit tests confirm correct loss/accuracy aggregation, prediction concatenation, and logging calls.
- **M9 - Explanation Module:** This module implements the subgraph explanation strategy. Unit tests cover internal score computation, prototype similarity matching, recursive backpropagation, and high-level explanation behavior.

Rationale for Non-Tested Modules. The following modules are not unit-tested either due to their indirect role, external origin, or verification through integration/system-level tests:

- M1 - Hardware-Hiding Module: This module abstracts disk I/O and memory operations. Its functionality is implicitly tested through file saving/loading operations in other modules, such as checkpoint storage and dataset caching.
- M2 - Configuration Module: This module only contains global constants and command-line configuration structures. Since it defines no active computation, it is not unit-tested but is covered implicitly through the modules that consume its parameters.
- M4 - Control Module: This module is the main program and serves as the orchestrator for all other modules. Its correctness is indirectly verified through the unit tests of the functional modules it calls.
- M7 - Model Module: This module imports standard GNN backbones from published codebases. These components are assumed correct and excluded from internal unit testing.

5.2 Tests for Functional Requirements

5.2.1 Input Format Module (M3)

This module corresponds to the data ingestion and preprocessing layer responsible for loading datasets, applying transformations, and preparing batched dataloaders for graph classification. The tests below follow the specifications outlined in the MIS, covering access programs such as load_dataset,

_get_dataloader, and include wrapper functions for MUTAG and other supported datasets. Each test is derived from either a normal usage pattern or a defined edge case. The goal is to verify expected input-output behavior and robust error handling.

1. test-M3-1: Dataset Loading Interface Test
   Type: Automatic, Functional
   Initial State: No dataset is currently loaded
   Input: Dataset name (e.g., MUTAG), dataset path, and training configuration
   Output: Dataset object with valid num_node_features and num_classes; a dictionary containing train, eval, and test DataLoaders
   Test Case Derivation: Verifies the behavior of load_dataset() and confirms correct metadata extraction and data loader construction.
   How test will be performed: Patch internal helpers (_get_dataset, _get_dataloader), call load_dataset(), and assert on returned structure and contents.

2. test-M3-2: Random Split Dataloader Construction Test
   Type: Automatic, Functional
   Initial State: A PyTorch Geometric dataset object is available
   Input: List of synthetic Data objects, split ratio [0.6, 0.2, 0.2]
   Output: DataLoader dictionary with train, eval, and test splits totaling original size
   Test Case Derivation: Ensures that randomized dataset splitting logic in _get_dataloader() produces reproducible and valid splits
   How test will be performed: Pass a dataset with batch size and split ratio into _get_dataloader() and assert that the returned dictionary contains keys "train", "eval", and "test".

3. test-M3-3: Predefined Split Index Test
   Type: Automatic, Functional
   Initial State: Dataset includes a supplement['split_indices'] attribute
   Input: Dataset with split indices tensor defining train, eval, and test members
   Output: Three DataLoaders corresponding to these index masks
   Test Case Derivation: Required for compatibility with datasets that ship with predefined splits.
   How test will be performed: Mock dataset with index labels and verify correct construction of DataLoaders using subset masks.

4. test-M3-4: Dataset Wrapper Invocation Test (MUTAG)
   Type: Automatic, Functional
   Initial State: No dataset loaded
   Input: Dataset name MUTAG and directory path
   Output: _MUTAGDataset object loaded from saved data.pt
   Test Case Derivation: Verifies correct dispatching from _get_dataset() to MUTAG-specific wrapper and confirms the dataset inherits InMemoryDataset
   How test will be performed: Patch torch.load and process(), create _MUTAGDataset, and validate class and internal attributes.

5. test-M3-5: Unsupported Dataset Handling Test
   Type: Automatic, Functional
   Initial State: No dataset selected
   Input: Dataset name not recognized by dispatch (e.g., 'FakeDataset')
   Output: NotImplementedError is raised
   Test Case Derivation: Confirms graceful failure when an invalid or unsupported dataset name is passed
   How test will be performed: Pass a fake name into _get_dataset() and check the correct exception is raised.

5.2.2 Training Module (M5)

The Training Module coordinates the warm-up, projection, and full training phases of the prototype-based GNN. It computes loss, handles optimizer updates, evaluates validation performance, manages prototype alignment, and triggers checkpoint saving. Unit tests are implemented to cover all internal access routines as specified in the MIS, and to isolate individual components of the training logic such as loss computation, dataset statistics, training mode configuration, and prototype behavior. The primary training loop itself is indirectly tested through system-level loss convergence (see 4.1.2).

1. test-M5-1: Total Loss Computation Test
   Type: Automatic, Functional
   Initial State: Dummy model with initialized prototype identity and parameters
   Input: Logits, class labels, minimum distances, loss function, and regularization weights
   Output: Scalar tensor representing the total loss

Test Case Derivation: Ensures correct aggregation of classification, cluster, separation, L1, and diversity losses.
How test will be performed: Generate synthetic logits and distances, and pass them into _compute_total_loss().

2. test-M5-2: Dataset Statistics Logging Test
   Type: Automatic, Functional
   Initial State: Dataset of graphs with fixed node and edge counts
   Input: Dataset list of PyG-compatible data objects
   Output: Printed statistics and log string
   Test Case Derivation: Logs dataset-level statistics for monitoring purposes.
   How test will be performed: Use capsys to capture standard output of _log_dataset_stats() and verify its format.

3. test-M5-3: Training Mode Configuration Test
   Type: Automatic, Functional
   Initial State: Model with trainable GNN layers and classifier head
   Input: warm_only=True and False flags
   Output: Parameters' requires_grad state adjusted accordingly
   Test Case Derivation: Distinguishes between warm-up and full joint training modes.
   How test will be performed: Call _set_training_mode() and ensure no exceptions and state changes occur.

4. test-M5-4: Prototype Projection Test
   Type: Automatic, Functional
   Initial State: Model and dataset with one-hot labels
   Input: Dataset, prototype vectors, and prototype-to-label mapping
   Output: Projected prototype vector aligned to closest explanation
   Test Case Derivation: Validates the projection of each prototype to the most similar real sample.
   How test will be performed: Patch get_explanation() and call _project_prototypes() to check for correct assignment and execution.

5. test-M5-5: Evaluation Routine Test
   Type: Automatic, Functional
   Initial State: Model in evaluation mode with known output predictions
   Input: Validation dataloader and loss function

Output: Mean loss and accuracy across batches
Test Case Derivation: Measures performance statistics for early stopping and model selection.
How test will be performed: Use a model with fixed logits and assert expected output keys in the result dictionary.

Additional Note: While the top-level train() function is not directly unit-tested, it is indirectly verified through system test T2 (see 4.1.2), which performs end-to-end training runs and confirms expected loss convergence over epochs. This test implicitly validates the integration of training components such as optimizer updates, prototype projection, and checkpoint logic.

5.2.3 Output Visualization Module (M6)

This module handles logging, model checkpoint saving, and visual explanation rendering for subgraph-based GNN interpretability. Unit tests have been designed to cover each public access routine defined in the MIS for this module, including standard operations for logging, model saving, and drawing subgraphs across supported datasets. The tests address both expected behavior and edge case handling (e.g., unsupported dataset types). The module's correctness is validated by asserting the file writes and method calls.

1. test-M6-1: Logging Append Test
   Type: Automatic, Functional
   Initial State: Log file is empty or preexisting
   Input: A single-line string (e.g., test log line)
   Output: The string is appended to the ./results/log/hyper_search log file with a newline character
   Test Case Derivation: Validates the basic logging utility used across training and inference scripts.
   How test will be performed: Patch open() and check that the file was opened in append mode and write() was called with the correct formatted input.

2. test-M6-2: Model Save with Best Checkpoint
   Type: Automatic, Functional
   Initial State: A model object is instantiated

Input: Checkpoint directory, epoch number, model, model name, accuracy value, and is_best=True flag
Output: A checkpoint file is saved, and a copy is made to the best checkpoint path
Test Case Derivation: Ensures that the best model is correctly tracked and persisted
How test will be performed: Patch torch.save and shutil.copy, then assert both were called correctly.

3. test-M6-3: Model Save without Best Checkpoint
   Type: Automatic, Functional
   Initial State: A model object is instantiated
   Input: Checkpoint directory, epoch number, model, model name, accuracy value, and is_best=False flag
   Output: A checkpoint file is saved, but no copy is made for best checkpoint
   Test Case Derivation: Validates that saving logic bypasses best checkpoint logic when not applicable
   How test will be performed: Patch torch.save and shutil.copy, and assert only the save call is invoked.

4. test-M6-4: Explanation Dispatcher Test (MUTAG)
   Type: Automatic, Functional
   Initial State: ExpPlot initialized with dataset name
   Input: A NetworkX graph, list of nodes to highlight, and a feature matrix
   Output: Call to _draw_molecule() subroutine with appropriate arguments
   Test Case Derivation: Ensures correct routing for molecule-based explanations
   How test will be performed: Patch _draw_molecule method and assert it is called with expected input.

5. test-M6-5: Subgraph Drawing Output Test
   Type: Automatic, Functional
   Initial State: ExpPlot object instantiated with a supported dataset name
   Input: A NetworkX graph and temporary output file path
   Output: An image file (e.g., PNG) is saved containing the visualized

subgraph
Test Case Derivation: Basic requirement that the visualization function executes without error
How test will be performed: Create a test graph, call _draw_subgraph(), and verify that the file exists at the output location.

6. test-M6-6: Dataset Routing Failure Test
   Type: Automatic, Functional
   Initial State: ExpPlot initialized with an unsupported dataset name
   Input: Any graph and a list of nodes
   Output: A NotImplementedError is raised
   Test Case Derivation: Prevents silent failures for unrecognized datasets in the drawing logic
   How test will be performed: Patch matplotlib.pyplot.savefig to suppress side effects and assert that an error is raised when drawing with an invalid dataset name.

5.2.4 Inference Module (M8)

This module encapsulates the inference-time behavior of the trained GNN model. It is responsible for computing predictions, evaluating performance on unseen data, and logging results. The primary access routine run_inference is verified using unit tests that validate its output structure, accuracy computation, loss aggregation, and logging side effects. Both functional and boundary conditions (e.g., single-batch datasets) are tested to ensure stability across usage scenarios defined in the MIS.

1. test-M8-1: Inference Execution and Output Test
   Type: Automatic, Functional
   Initial State: Trained model and loss function loaded
   Input: Evaluation dataloader, model object, and loss function
   Output: Dictionary with loss and accuracy, prediction logits and probabilities
   Test Case Derivation: Ensures the function executes end-to-end and returns all expected fields.
   How test will be performed: Pass a dataloader and model to the inference routine and verify return structure.

2. test-M8-2: Accuracy Calculation Verification
   Type: Automatic, Functional
   Initial State: Model returns known logits and true labels
   Input: Model with fixed output and corresponding dataloader
   Output: Correct accuracy calculation in result dictionary
   Test Case Derivation: Verifies the correctness of the accuracy logic based on class match count.
   How test will be performed: Use controlled outputs from a model to assert expected accuracy values.

3. test-M8-3: Inference Logging Test
   Type: Automatic, Functional
   Initial State: Logging file initialized or exists
   Input: Same as above
   Output: Record appended to log file
   Test Case Derivation: The system is expected to log inference results consistently.
   How test will be performed: Patch the logging function and verify it is invoked with correctly formatted result.

4. test-M8-4: Batch Aggregation Consistency Test
   Type: Automatic, Functional
   Initial State: Model returns outputs in multiple batches
   Input: Multi-batch dataloader and inference model
   Output: Concatenated arrays for predictions and probabilities
   Test Case Derivation: All batches should be merged for final output consistency.
   How test will be performed: Provide two or more batches and check that final arrays match the total batch size.

5.2.5 Explanation Module (M9)

The Explanation Module is responsible for identifying and visualizing the most relevant subgraph for a given GNN prediction using a Monte Carlo Tree Search (MCTS)–based strategy. The associated unit tests ensure correctness of explanation scores, rollout behavior, subgraph selection, and visualization triggering. Tests are selected to exercise both the black-box interface and key white-box mechanisms described in the MIS, such as prototype similarity scoring, node expansion, and recursive score propagation.

1. test-M9-1: MCTS Node Scoring Test
   Type: Automatic, Functional
   Initial State: Node created with initialized state parameters
   Input: A coalition and dummy graph
   Output: Valid computed Q and U scores
   Test Case Derivation: Verifies the scoring behavior of tree nodes used in exploration phase
   How test will be performed: Instantiate a node and validate returned Q and U values

2. test-M9-2: Prototype Similarity Test
   Type: Automatic, Functional
   Initial State: Model in evaluation mode with fixed embedding output
   Input: Coalition and prototype tensor
   Output: A scalar similarity score
   Test Case Derivation: Ensures prototype-sample similarity is correctly measured
   How test will be performed: Use a dummy GNN to compute the similarity for a given coalition

3. test-M9-3: MCTS Scoring Utility Test
   Type: Automatic, Functional
   Initial State: Multiple candidate nodes with varying priors
   Input: List of MCTS nodes and scoring function
   Output: List of computed scores
   Test Case Derivation: Verifies score selection logic in exploration step
   How test will be performed: Run scoring on multiple nodes and check ordering and values

4. test-M9-4: MCTS Rollout Test
   Type: Automatic, Functional
   Initial State: Root node initialized with full coalition
   Input: Graph and scoring function
   Output: Updated node visit counts and scores
   Test Case Derivation: Validates recursive search and backpropagation logic
   How test will be performed: Perform a rollout and assert structural updates

5. test-M9-5: Explanation Function Test
   Type: Automatic, Functional
   Initial State: Dummy GNN and graph sample
   Input: Graph data, model, and prototype
   Output: Most relevant subgraph node list, score, and embedding
   Test Case Derivation: Checks output consistency for the high-level explanation function
   How test will be performed: Call explanation API and verify output types and values

6. test-M9-6: MCTS Rollout Backpropagation Test
   Type: Automatic, Functional
   Initial State: Root with one child node with initialized score
   Input: Score function
   Output: Updated statistics at root node
   Test Case Derivation: Verifies that child scores correctly propagate up the tree
   How test will be performed: Trigger rollout and validate state updates on the parent

5.3 Tests for Nonfunctional Requirements

Unit testing of nonfunctional requirements is considered out of scope for this project. These requirements will primarily be evaluated through the usability survey in Section 6.1.

5.4 Traceability Between Test Cases and Modules

This section provides evidence that all modules from the Module Hierarchy (Section 5) have been evaluated through targeted test cases where applicable. Each test case corresponds to a module or group of access routines within the module's responsibility.

Note: Modules M1, M2, M4, and M7 are not directly unit-tested due to the following reasons:

- **M1** (Hardware-Hiding Module): Tested implicitly through I/O in M6 and system test T1.

	M1	M2	M3	M4	M5	M6	M7	M8	M9
test-M3-1			X	X					
test-M3-2			X	X					
test-M3-3			X	X					
test-M3-4			X	X					
test-M3-5			X	X					
test-M6-1	X			X		X			
test-M6-2	X			X		X			
test-M6-3	X			X		X			
test-M6-4				X		X			
test-M6-5				X		X			
test-M6-6				X		X			
test-M5-1				X	X				
test-M5-2				X	X				
test-M5-3				X	X				
test-M5-4				X	X				
test-M5-5				X	X				
test-M5-6				X	X				
test-M8-1				X				X	
test-M8-2				X				X	
test-M8-3				X				X	
test-M8-4				X				X	
test-M9-1				X					X
test-M9-2				X					X
test-M9-3				X					X
test-M9-4				X					X
test-M9-5				X					X
test-M9-6				X					X

Table 3: Traceability Matrix of Test Cases and Modules

- **M2** (Configuration Module): No computational logic; parameters are exercised via other modules.
- **M4** (Control Module): Acts as the main orchestrator; validated indirectly via end-to-end system tests.
- **M7** (Model Module): Uses standard GNN backbones from existing implementations and is assumed correct.

References

Asim Kumar Debnath, Rosa L. Lopez de Compadre, Gargi Debnath, Alan J. Shusterman, and Corwin Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds: Correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry*, 34(2):786–797, February 1991. doi: 10.1021/jm00106a046.

Yuanqi Xue. Software requirements specification (srs) for re-protgnn, 2024. URL https://github.com/Yuanqi-X/Re-ProtGNN/blob/main/docs/SRS/SRS.pdf. Accessed: 2025-02-24.

Yuanqi Xue. Module guide (mg) for re-protgnn, 2025a. URL https://github.com/Yuanqi-X/Re-ProtGNN/blob/main/docs/Design/SoftArchitecture/MG.pdf. Accessed: 2025-04-13.

Yuanqi Xue. Mg checklist for re-protgnn, 2025b. URL https://github.com/Yuanqi-X/Re-ProtGNN/blob/main/docs/Checklists/MG-Checklist.pdf. Accessed: 2025-04-13.

Yuanqi Xue. Module interface specification (mis) for re-protgnn, 2025c. URL https://github.com/Yuanqi-X/Re-ProtGNN/blob/main/docs/Design/SoftDetailedDes/MIS.pdf. Accessed: 2025-04-13.

Yuanqi Xue. Srs checklist for re-protgnn, 2025d. URL https://github.com/Yuanqi-X/Re-ProtGNN/blob/main/docs/Checklists/SRS-Checklist.pdf. Accessed: 2025-04-13.

Yuanqi Xue. Verification and validation (v&v) plan for re-protgnn, 2025e. URL https://github.com/Yuanqi-X/Re-ProtGNN/blob/main/docs/VnVPlan/VnVPlan.pdf. Accessed: 2025-04-13.

Yuanqi Xue. V&v checklist for re-protgnn, 2025f. URL https://github.com/Yuanqi-X/Re-ProtGNN/blob/main/docs/Checklists/VnV-Checklist.pdf. Accessed: 2025-04-13.

Yuanqi Xue. Problem statement and goals for re-protgnn, 2025g. URL https://github.com/Yuanqi-X/Re-ProtGNN/blob/main/docs/ProblemStatementAndGoals/ProblemStatement.pdf. Accessed: February 6, 2025.

6 Appendix

6.1 Usability Survey Questions?

Please rate the following statements on a scale from 1 (Strongly Disagree) to 5 (Strongly Agree):

1. The system was easy to learn and use.
2. The interface is clear and intuitive.
3. The system responded in a reasonable amount of time.
4. I felt confident using the system without needing extra help.
5. Error messages, if any, were clear and helpful.
6. I am satisfied with my overall experience using the system.

Optional Open-ended Questions:

1. What did you like most about the system?
2. What improvements would you suggest?