System Verification and Validation Plan for Model–to–Brain Alignment for Speech Recognition

Xiao Shao

April 21, 2026

Revision History

Date	Version	Notes
Feb 10, 2026	1.0	Initial draft
Feb 13, 2026	1.1	Adjust according to feedback
March 29, 2026	1.2	Adjust according to feedback
Apr 21, 2026	1.3	Adjust according to smith's feedback

Contents

1 Symbols, Abbreviations, and Acronyms

Identical to the same section in SRS document (Shao, 2026).

This document presents the Verification and Validation (V&V) plan for Model–to–Brain Alignment for Speech Recognition. It explains how the software and related artifacts will be reviewed, tested, and evaluated to provide confidence that the implemented pipeline satisfies its documented requirements and is suitable for its intended use.

2 General Information

2.1 Summary

This project is for comparing candidate speech representations by how well they predict neural responses during speech recognition. Given speech stimuli and corresponding MEG recordings, the system constructs time-aligned predictors from multiple sources, fits ridge-regularized multivariate Temporal Response Function (mTRF) encoding models over a specified lag window, and produces quantitative scores and summaries to support reproducible scientific interpretation.

2.2 Objectives

The primary objectives of this Verification and Validation (V&V) plan are to:

- Build confidence that correctly performs the end-to-end workflow specified in the SRS.
- Demonstrate that comparisons across predictors are fair and reproducible under a consistent protocol.
- Provide evidence that the produced outputs are traceable and inspectable, enabling experiments to be repeated and extended with minimal ambiguity.

The following objectives are explicitly out of scope for this project:

- **Biological validity claims:** Model–to–Brain Alignment for Speech Recognition will not validate neuroscientific claims beyond what is implied by predictive performance. In particular, this project will not claim that higher prediction performance implies a true mechanistic explanation of the brain.

- **Formal verification:** This project will not provide formal proofs of correctness. The project resources are insufficient for a formal-methods effort.
- **Independent verification of external libraries:** This project assume that widely used scientific libraries (NumPy/SciPy, PyTorch, MNE-Python, Eelbrain) have been verified by their development communities.
- **Clinical-grade security:** The system is research software. Security hardening and clinical deployment requirements are out of scope.

2.3 Extras

- **Code walkthrough:** A structured walkthrough session will be conducted using the implementation. The walkthrough will examine whether each functional requirement is mapped to the intended modules, instance models, and verification activities. It will also inspect the consistency between the implementation, planned tests, and documented assumptions.
- **User manual:** A short user manual will be provided with environment setup instructions, required dependencies, configuration examples, and example commands for running one end-to-end experiment, including model training, predictor extraction, TRF fitting, and score computation.

2.4 Relevant Documentation

- Software Requirements Specification (SRS) Available at: SRS.
- Module Guide (MG). Available at: MG.
- Module Interface Specification (MIS). Available at: MIS.

3 Plan

This section describes the people and processes used to verify and validate Model–to–Brain Alignment for Speech Recognition. It covers document reviews, implementation verification, automated tools, and the approach to

Member	Role	Responsibilities
Xiao Shao	Developer	Produce the VnV Plan and refine based on feedback
Dr. Spencer Smith	Course Instructor	Provides feedback on engineering principles
Dr. Christian Brodbeck	Domain Expert	Provides domain-specific information on requirements and validation
Yanyu Wu	Reviewer	Provides feedback on VnV plan

Table 1: Verification and Validation Team

validating that the software meets stakeholder needs for reproducible scientific comparison.

3.1 Verification and Validation Team

3.2 SRS Verification

SRS verification will be conducted through a structured review process:

- **Checklist-based inspection:** The author will inspect the SRS using the course SRS checklist and a project-specific checklist focused on traceability between assumptions, theoretical models, instance models, data definitions, and requirements. Checklist link: Course SRS Checklist.
- **Peer review:** A classmate reviewer will review the SRS using the same checklist and written review instructions on Github. The review will focus on missing assumptions, ambiguous terminology, inconsistent references, and mismatches between requirements and instance models. Review comments will be recorded in the Github as issues or written review notes. Review record: SRS Review Comments.
- **Supervisor review:** The SRS will be discussed in a scheduled walkthrough meeting with the supervisor or instructor. The meeting will be structured around the problem scope, assumptions, requirements, theoretical and instance models, and traceability matrices. Action items from the meeting will be recorded and used to revise the document and framework. Meeting notes or summary: Supervisor Review Notes.

3.3 Design Verification

A classmate reviewer will review the MG and MIS using the corresponding course checklists and reviewer instructions. The review will focus on whether the module decomposition, module interfaces, and information hiding decisions are consistent with the SRS and with the planned implementation. Review comments will be recorded in the Github. MG review record: MG Review Comments. MIS review record: MIS Review Comments.

3.4 Verification and Validation Plan Verification

A classmate reviewer will review the VnV plan using the course checklist and the reviewer instructions provided by the author. The review will focus on whether each requirement and correctness property is mapped to a concrete verification or validation activity, and whether the proposed tests are specific and reproducible. Review comments will be recorded in the Github. VnV review record: VnV Review Comments.

3.5 Implementation Verification

- **Unit testing:** Automated tests are used for core modules (alignment, lagged design matrix construction, ridge fitting, scoring, and artifact logging). These tests cover both normal behavior and edge cases, including shape mismatches, invalid inputs, and missing or NaN values.
- **Integration testing:** End-to-end pipeline tests on Librispeech dataset are used to verify correct interaction among modules, including data loading, predictor construction, temporal alignment, model fitting, scoring, and result generation.
- **Requirement-based verification:** System-level checks are used to confirm that the implementation satisfies the intended functional behavior described in the SRS. For example, tests verify that all predictors specified in an experiment are evaluated under the same protocol, that cross-validated scores are aggregated as specified, that baseline-referenced gains are computed and recorded, and that invalid inputs are rejected before model fitting.

- **Reproducibility and regression checks:** Repeated runs under the same configuration are compared to confirm stable behavior, and regression tests are used to detect unintended changes in outputs after implementation updates.

3.6 Automated Testing and Verification Tools

- **pytest** for unit and integration tests, with markers for slow tests.
- **GitHub Actions** to automates CI/CD workflows to run tests, checks, and deployments on every push or pull request.
- **flake8** for linting and black for formatting to enforce coding standards.
- **Continuous Integration** on the repository to run tests and checks on each merge request, ensuring regressions are caught early.

3.7 Software Validation

Out of scope for this project.

4 System Tests

This section lists system-level tests that collectively cover the functional and nonfunctional requirements in the SRS. Tests are grouped by major workflow areas: input/validation, predictor construction and alignment, TRF fitting, evaluation and gain computation.

Definition of Clean Environment A clean environment means that the test begins from a fresh execution state with no residual files, caches, in-memory objects, or configuration side effects from previous runs, except for the inputs and configuration explicitly specified in the current test case.

4.1 Tests for Functional Requirements

The functional tests below cover the end-to-end workflow implied by the SRS requirements.

4.1.1 Input and Configuration Validation

Input schema and basic validity checks

1. FR-01 (Invalid MEG sampling frequency)
   Control: Automatic
   Initial State: Clean environment; a valid MEG metadata record extracted from an Appleseed MEG file is available (for example, via a test fixture or a copied header from one valid file).
   Input: Start from a valid Appleseed MEG metadata record and set the sampling frequency field `fs` to an invalid value with `fs = 0` (or any `fs < 0`).
   Output: The system rejects the run and reports an `InvalidConfigurationError` (or equivalent validation failure) that identifies the MEG sampling frequency field and states that the value must be strictly positive.
   Test Case Derivation: R1
   How test will be performed: In pytest, load a valid Appleseed-derived metadata fixture, modify the field `fs` to `0`, call the input or metadata validation function, and assert that validation fails with an error message containing both the field name `fs` and the constraint that the sampling frequency must be positive.

2. FR-02 (Unsupported deep model type)
   Control: Automatic
   Initial State: Clean environment; a valid model configuration is loaded.
   Input: Start from a valid model configuration and set the field `model.type` to a specific unsupported identifier, such as `"unsupported_model"`.
   Output: The system rejects the configuration and reports that the specified model type is unsupported. The error message shall also list the supported model types (for example, `RNN`, `LSTM`, `GRU`, `Transformer`).
   Test Case Derivation: R2
   How test will be performed: In pytest, load a valid model configuration, change `model.type` to `"unsupported_model"`, call the model configuration validation function, and assert that validation fails with an error message that includes the invalid identifier and the supported model type list.

3. FR-03 (Invalid model architecture hyperparameters)
   Control: Automatic

Initial State: Clean environment; a valid model configuration is loaded.
Input: Create invalid configuration variants from the valid model configuration by setting one hyperparameter at a time to an invalid value: `num_layers = 0`, `hidden_size = 0`, and `dropout = 1.2`.
Output: For each invalid configuration, the system rejects the configuration and reports a validation error that identifies the invalid hyperparameter name and states the violated constraint (for example, `num_layers > 0`, `hidden_size > 0`, or `dropout` $\in [0, 1)$).
Test Case Derivation: R2
How test will be performed: In pytest, parameterize the three invalid configuration variants, call the model configuration validation function for each case, and assert that validation fails with an error message mentioning the corresponding hyperparameter name and constraint.

4. FR-04 (Model input feature dimension mismatch)
   Control: Automatic
   Initial State: Clean environment; a valid model configuration is loaded, and the configured model expects input tensors with feature dimension D_{exp} (for example, the number of acoustic features or gammatone channels specified in the configuration).
   Input: Construct a small synthetic input batch whose last dimension is $D_{\text{obs}} \neq D_{\text{exp}}$; for example, if the model expects $D_{\text{exp}} = 64$, provide an input tensor with feature dimension $D_{\text{obs}} = 80$.
   Output: The system rejects training or representation extraction and reports a dimension-mismatch error identifying both the expected feature dimension and the observed feature dimension.
   Test Case Derivation: R2
   How test will be performed: In pytest, load a valid model configuration, construct a synthetic batch with mismatched feature dimension, call the model input validation or forward-pass entry point, and assert that an exception is raised before any optimizer step, with an error message containing both D_{exp} and D_{obs}.

5. FR-05 (LibriSpeech manifest integrity)
   Control: Automatic
   Initial State: Clean environment; a valid LibriSpeech-style manifest or metadata table is available, and the dataset root path is configured (for example, `train-clean-100`).

Input: Create a test manifest derived from the valid manifest in which one row references a missing audio file or a missing transcript file. The modified row identifier shall be recorded in the test input.
Output: The system rejects dataset loading and reports the missing file path together with the identifier of the manifest row that contains the invalid reference.
Test Case Derivation: R2
How test will be performed: In pytest, load the modified manifest, call the dataset-loading or manifest-validation function, and assert that loading fails with an error message containing the missing path and the corresponding row identifier.

6. FR-06 (Incompatible predictor time axis for alignment)
   Control: Automatic
   Initial State: Clean environment; MEG data are loaded with a known MEG sampling rate and time grid.
   Input: Provide a predictor time series whose time axis is incompatible with the MEG time grid; for example, omit the predictor timestamps entirely, provide a non-monotonic predictor time axis, or provide timestamps whose covered time range does not overlap the MEG segment to be aligned.
   Output: The system rejects alignment and reports a validation error identifying the predictor time axis and the MEG time grid as incompatible for alignment.
   Test Case Derivation: R4
   How test will be performed: In pytest, create a small synthetic MEG time grid and a predictor with an incompatible time axis, call the predictor-alignment function, and assert that alignment fails with an error message describing the specific incompatibility.

7. FR-07 (Inconsistent evaluation protocol across predictors)
   Control: Automatic
   Initial State: Clean environment; at least two predictor definitions are configured (for example, a gammatone baseline and one hidden-state predictor).
   Input: Configure the two predictors with different evaluation protocol settings; for example, assign different lag windows, different lag discretizations, different cross-validation partitions, or different regu-

larization grids.
Output: The system rejects the evaluation run and reports which protocol fields differ across predictors.
Test Case Derivation: R7
How test will be performed: In pytest, define two predictors, intentionally set one protocol field to a conflicting value for one predictor, call the evaluation protocol validation function, and assert that validation fails with an error message listing the conflicting protocol field names.

8. FR-08 (Encoding score computation returns invalid or missing results)
Control: Automatic
Initial State: Clean environment; one predictor matrix and one MEG response matrix are available on the same time grid.
Input: Construct a test case in which the score is undefined or invalid; for example, (a) insert a `NaN` value into the predictor matrix, or (b) provide a constant response signal that makes Pearson correlation undefined.
Output: The system rejects scoring and reports a scoring error identifying the cause of invalidity, such as the presence of `NaN` values in the predictor matrix or undefined correlation due to a constant response signal.
Test Case Derivation: R7
How test will be performed: In pytest, construct small synthetic matrices for the predictor and response, evaluate the scoring function under each invalid case, and assert that scoring fails with an error message indicating the specific cause of invalidity.

4.2 Tests for Nonfunctional Requirements

4.2.1 Usability

Usability survey acceptance test

1. NFR-01
Type: Manual
Initial State: An alpha version of Model–to–Brain Alignment for Speech Recognition is available; a user group consisting of graduate students or researchers familiar with speech/MEG analysis is selected.

Input/Condition: Participants complete a short end-to-end task (configure paths, run one training-or-load step, extract representations as predictors, fit mTRF, and open the final comparison summary) and then complete a usability survey questionnaire.
Output/Result: At least 80% of respondents rate the software as easy to use.
How test will be performed: Collect questionnaire results; compute the percentage of "easy to use" responses and verify it is larger than 80%.

4.2.2 Maintainability

Likely-change effort threshold test

1. NFR-02
   Type: Automatic, Manual
   Initial State: Baseline development time is recorded for the original implementation; the codebase is under version control and includes a basic test suite.
   Input/Condition: Select one likely change (e.g., adding one new deep model type, or adding one new predictor extraction option such as an additional layer output) and implement it by a developer with relevant domain knowledge.
   Output/Result: The measured effort to implement the chosen likely change is less than 10 lines of the code, and existing tests still pass.
   How test will be performed: Track engineering time using commits/issues; compare total effort against the recorded baseline and confirm it satisfies the threshold.

4.2.3 Portability

Cross-platform installation and execution test

1. NFR-03
   Type: Manual
   Initial State: Clean Windows, macOS, and Linux environments prepared.
   Input/Condition: Install Model–to–Brain Alignment for Speech Recognition and run a small smoke-test workflow on each OS using a bundled demo configuration and a small subset of Librispeech.

Output/Result: The pipeline installs and runs successfully on Windows, macOS, and Linux and produces the expected output files for the workflow.
How test will be performed: Create different workflows for each OS on GitHub Actions (or run the same documented steps manually if CI is unavailable) and record pass/fail logs.

4.2.4 Reproducibility

Repeat-run consistency test

1. NFR-04
   Type: Automatic
   Initial State: A fixed configuration file is available; the run uses a fixed random seed; the same input data are used; deterministic execution settings are enabled where supported.
   Input/Condition: Run the same end-to-end workflow twice without changing the configuration, input data, code version, or random seed. The workflow includes model training (or model loading), predictor extraction, mTRF fitting, score computation, and report generation.
   Output/Result: The two runs shall produce: (a) identical output artifact names and directory structure, and (b) numerically matching summary results within a fixed tolerance. In particular, the reported cross-validated scores, baseline-referenced improvements, and other scalar summary metrics shall differ by no more than $\epsilon = 10^{-8}$.
   How test will be performed: In pytest, execute the pipeline twice using the same configuration and seed. Compare the lists of generated output files, and compare the numeric fields in the generated summary files using an absolute tolerance of 10^{-8}. The test passes only if the artifact structure is the same and all compared summary values fall within the specified tolerance.

4.2.5 Performance

End-to-end runtime baseline benchmark

1. NFR-05
   Type: Automatic
   Initial State: A fixed benchmark configuration is available, including a

short audio segment, a small MEG segment, one candidate predictor, and a minimal cross-validation setting.
Input/Condition: Run the full workflow on the benchmark configuration, including predictor extraction, predictor alignment, design matrix construction, TRF fitting, score computation, and summary generation.
Output/Result: The workflow shall complete successfully and produce the expected output artifacts, including: (a) an aligned predictor matrix file, (b) a fitted TRF result file, (c) a score summary file, and (d) a timing log. The run shall terminate without memory exhaustion or runtime failure.
How test will be performed: Record wall-clock runtime and peak memory usage for the benchmark run. Verify that all expected output artifacts are created and that the run completes successfully. Store the timing log together with the output artifacts for later comparison.

4.3 Traceability Between Test Cases and Requirements

Requir.	**Test Cases**
R1	FR-01
R2	FR-02
R3	FR-03
R4	FR-06
R5	FR-04
R6	FR-05
R7	FR-07, FR-08
NFR1	NFR-01
NFR2	NFR-02
NFR3	NFR-03
NFR4	NFR-04
NFR5	NFR-05

Table 2: Traceability Between Requirements and Test Cases

5 Unit Test Description

5.1 Unit Testing Scope

5.2 Tests for Nonfunctional Requirements

5.3 Traceability Between Test Cases and Modules

References

X. Shao. System requirements specification. https://github.com/..., 2026.