COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

UNIVERSITY MASTER'S DEGREE IN TELECOMMUNICATIONS TECHNOLOGIES ENGINEERING

MASTER'S THESIS

MULTI-LAYER VERIFICATION HARNESS FOR LLM-GENERATED CODE PATCHES

Author: Ignacio Hernández Bas

Director: Furong Huang

Madrid

Junio de 2026

Declaration of Authorship and Originality

I hereby declare, under my responsibility, that the project presented with the title **: A Multi-Layer Verification Harness for LLM-Generated Code Patches,** submitted to the School of Engineering (ICAI) at Universidad Pontificia Comillas during the academic year **2025/2026**, is my own original work and has not been previously submitted for any other academic purpose. This project does not constitute plagiarism, either in whole or in part, and all information taken from other sources has been properly referenced.

Declaration on the Use of Artificial Intelligence[1]

I hereby declare, under my responsibility, that (please select the appropriate option):

☐ I have not used Artificial Intelligence in the preparation of this document.

X I have used Artificial Intelligence in the preparation of this document and/or Annex B, always in accordance with the conditions permitted by Universidad Pontificia Comillas, specifically applying Level 2 of the Evaluation Scale defined by Perkins et al. (2024): "AI may be used for pre-task activities such as brainstorming, outlining, and initial research. This level focuses on the use of AI for planning, synthesis, and idea generation, while assessments must emphasize the ability to independently develop and refine these ideas. " Specifically, Artificial Intelligence has been used for:

Brainstorming and Planning: Assistance in drafting the initial structure of the thesis and planning the organization of its sections.

Research and Literature Review: Support in exploring related work and identifying relevant research directions.

Technological Recommendations: Guidance in evaluating and comparing potential technologies, libraries, and tools suitable for the project.

Tool Usage and Documentation: Assistance in understanding and applying technical documentation, including the use of infrastructure tools such as cluster environments (e.g., Slurm) and container-based execution.

Signed (alumno): Ignacio Hernández Bas

Fecha: 30-03-2026

[1] This declaration refers to the use of generative Artificial Intelligence in the preparation of the Project documents (Annex B and the thesis report). It does not apply to projects where Artificial Intelligence is inherently part of the work itself (e.g., applications of machine learning techniques, neural networks, data analysis, etc.).

Project Submittal Authorization

Project director	Project co-director
	
Signed: Dr Furong Huang	Signed:
Date: 16-30-2026	Date:

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

UNIVERSITY MASTER'S DEGREE IN TELECOMMUNICATIONS TECHNOLOGIES ENGINEERING

MASTER'S THESIS

MULTI-LAYER VERIFICATION HARNESS FOR LLM-GENERATED CODE PATCHES

Author: Ignacio Hernández Bas

Director: Furong Huang

Madrid

Junio de 2026

MULTI-LAYER VERIFICATION HARNESS FOR LLM-GENERATED CODE PATCHES

Author: Ignacio Hernández Bas
Director: Furong Huang
Collaborating Entity: University of Maryland

ABSTRACT

This thesis proposes a multi-layer verification harness to evaluate LLM-generated Python code patches using signals beyond pre-existing unit tests. The pipeline combines patch-scoped **static analysis**, including a composite Static Quality Index derived from multiple linters, containerized **dynamic execution** of repository tests, and a **semantic layer** that extracts structured behavioural claims from issue descriptions and synthesises discriminative pytest tests via an agentic refinement loop. These tests are validated by requiring failure on the buggy version and success on the reference, or *gold*, fix.

On SWE-bench Lite, the static layer is applied to 300 gold patches, while the dynamic and semantic experiments are conducted on 18 instances with available execution images. Across all model pairings, 37.4% of claim tests become discriminative; the best pairing reaches 64.3% (18/28). Candidate-patch experiments show that some patches passing the benchmark suite can still diverge from the reference behaviour. This highlights both the usefulness of the semantic layer for detecting differences not covered by existing tests and a limitation of reference-based validation: generated tests may become partially anchored to the concrete behaviour of the *gold* patch.

Keywords: LLM-Generated Code Patches, Multi-Layer Verification, Patch Correctness Evaluation, Semantic Verification, Differential Testing, SWE-bench Lite

1. Introduction

Recent advances in Large Language Models (LLMs) have significantly improved the ability to generate code patches for real-world software issues [1]. While benchmarks such as SWE-bench [2] demonstrate that LLMs can produce patches that pass existing unit tests, these evaluations remain limited to test-suite adequacy and do not guarantee alignment with the intended behavior described in issue reports. Recent studies further show that patches classified as solved may still be semantically incorrect [3].

This project addresses this limitation by proposing a verification framework that extends beyond test-suite validation. Instead of relying solely on existing tests, the system introduces additional verification layers that assess structural correctness, execution behavior, and alignment with the natural-language specification of the problem. The goal is to provide a more comprehensive evaluation signal for LLM-generated patches.

2. Project Definition

The core problem addressed in this work is the gap between *test-suite correctness* and *semantic correctness*. Formally, given a repository, a buggy version, and a candidate patch, the objective is to determine whether the patch satisfies the intended behavior described in the issue.

The project is based on the following verification hypothesis:

> A multi-layer verification pipeline combining static, dynamic, and semantic analysis provides a more informative and reliable evaluation of patch correctness than test-suite-based validation alone.

The system is evaluated using SWE-bench Lite [2], which provides real-world software issues, corresponding patches, and execution environments. Static analysis is applied to the full dataset, while dynamic and semantic verification are evaluated on a subset of instances due to infrastructure constraints.

3. System Description

The proposed system is structured as a multi-layer verification harness (see Figure 1) composed of three sequential modules:

1. **Static Verification Layer:** This layer performs fast, deterministic analysis of candidate patches without executing the code. It includes syntax validation, structural analysis, and a composite Static Quality Index (SQI) based on tools such as Pylint, Radon, Flake8, Mypy, and Bandit. Its purpose is to filter out structurally invalid or low-quality patches early in the pipeline.

2. **Dynamic Verification Layer:** This layer executes the repository's original unit tests in a containerized environment. It verifies that the patch resolves the targeted bug (*fail_to_pass*) without introducing regressions (*pass_to_pass*). This stage reproduces the standard evaluation protocol used in SWE-bench [2].

3. **Semantic Verification Layer:** This layer introduces behavioral validation derived from the issue description. It extracts structured claims using an LLM, grounds them to relevant code elements, and generates executable tests through an agentic loop. These tests are validated using a discriminative execution scheme (BUG fails, GOLD passes), inspired by prior work on differential and regression test generation [4, 5]. Additionally, recent work has shown that LLMs can be used to generate tests directly from issue descriptions [6].

The system follows an iterative refinement process in which test generation is guided by execution feedback, allowing progressive improvement of test quality.

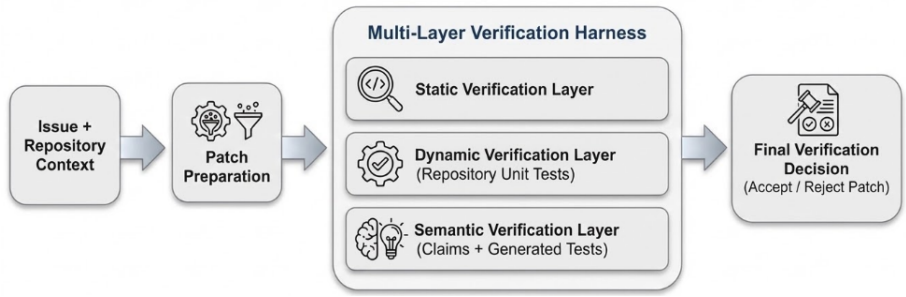

Figure 1: Verification Pipeline Overview

4. Results

The static verification layer was successfully applied to all 300 instances in SWE-bench Lite, providing consistent structural and quality assessments across repositories.

The dynamic verification layer was evaluated on a subset of 18 instances with available execution environments, achieving a high success rate in validating gold patches and confirming correct bug resolution without regressions.

The semantic verification layer demonstrated the ability to generate discriminative tests that capture behavioral intent. Experimental results show that these tests can identify semantic differences among patches that pass the original test suite, a known limitation in automated program repair [5]. In particular, a significant proportion of candidate patches aligned with the reference implementation were found to be consistent with the reference implementation, while others were identified as overconstrained or semantically divergent.

Layer	**Instances**	**Pass Rate**	**Verification Signal**
Static	300	100% syntax valid, 87.7% Good+ SQI	Structural quality
Dynamic	18	17/18 (94.4%) resolved	Test-suite alignment
Semantic	18	Up to 64.3% discriminative (best config)	Reference-aligned behavioral validation

Table 1: Cross-layer summary of the verification pipeline. Each layer provides a complementary verification signal, progressively increasing behavioral specificity while reducing coverage.

Figure 4 provides a detailed comparison of the discriminative rate across different model combinations for claim extraction and test generation. The results highlight a strong dependence on the chosen model pairing, with performance varying significantly across configurations.

The best-performing combination is obtained using Qwen3-32B for both claim extraction and test generation, achieving a discriminative rate of 64.3% (18/28). In contrast, other combinations

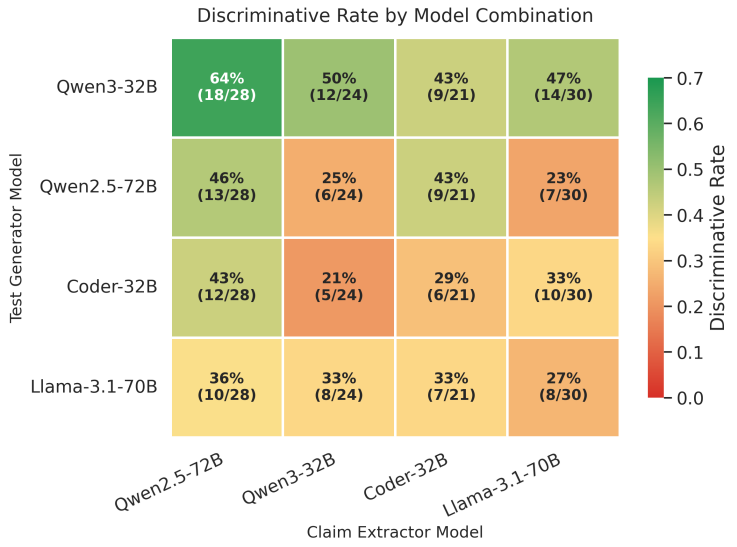

Figure 2: Discriminative rate across model combinations (test generator vs claim extractor).

show substantially lower performance, in some cases below 25%, indicating that both stages of the semantic pipeline contribute critically to overall effectiveness.

These results suggest that the quality of the semantic verification layer is not determined solely by the test generation model, but also by the structure and relevance of the extracted claims. Strong claim extraction enables the generation of more discriminative and behaviorally meaningful tests, while weaker claims limit the effectiveness of the entire pipeline.

Overall, the observed variability reinforces the importance of jointly optimizing both components of the semantic layer rather than treating them as independent modules, as performance depends critically on the interaction between claim extraction and test generation.

However, the results also reveal an important phenomenon: the generated tests tend to align with the behavior of the reference (gold) patch. This effect, referred to as *reference-based anchoring*, highlights that validation is inherently relative to the reference implementation rather than a complete specification of the problem.

Importantly, passing the dynamic layer does not guarantee semantic correctness, reinforcing the need for behavioral validation beyond test-suite adequacy. These results illustrate that each verification layer provides a complementary signal, progressively increasing the specificity of patch validation while reducing coverage due to execution constraints.

Overall, the multi-layer approach provides a richer evaluation signal compared to test-suite-only validation, improving the detection of subtle behavioral differences.

5. Conclusions

This project demonstrates that combining static, dynamic, and semantic verification enables a more comprehensive assessment of LLM-generated code patches. The semantic layer, in particular, extends traditional evaluation by incorporating behavioral intent derived from natural-language issue descriptions.

The results confirm the initial hypothesis: multi-layer verification improves the ability to detect incorrect or incomplete patches beyond what is captured by existing test suites. However, the study also identifies key limitations, especially the dependence on a reference implementation for validation, which may bias the evaluation process.

Future work should focus on reducing this dependency by exploring alternative validation strategies, such as specification-based testing, multiple reference implementations, or human-in-the-loop evaluation. Additionally, scaling the semantic layer to larger datasets and improving claim extraction robustness remain important challenges.

In addition to its empirical findings, this work contributes a modular and extensible framework for multi-layer patch verification, which can serve as a foundation for future research in LLM-based software engineering evaluation.

6. References

[1] Juyong Jiang et al. "A Survey on Large Language Models for Code Generation". In: *ACM Transactions on Software Engineering and Methodology* 35.2 (Jan. 2026), pp. 1–72. DOI: 10.1145/3747588.

[2] Carlos E. Jimenez et al. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" In: *The Twelfth International Conference on Learning Representations (ICLR)*. 2024. URL: https://openreview.net/forum?id=VTF8yNQM66.

[3] You Wang, Michael Pradel, and Zhongxin Liu. "Are "Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study". In: *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*. 2025. URL: https://arxiv.org/abs/2503.15223.

[4] Kunal Taneja and Tao Xie. "DiffGen: Automated Regression Unit-Test Generation". In: *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2008, pp. 407–410. DOI: 10.1109/ASE.2008.60.

[5] Qi Xin and Steven P. Reiss. "Identifying Test-Suite-Overfitted Patches through Test Case Generation". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2017, pp. 226–236. DOI: 10.1145/3092703.3092718.

[6] Sungmin Kang, Juyeon Yoon, and Shin Yoo. "Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction". In: *ICSE*. 2023. URL: https://arxiv.org/abs/2209.11515.

EVALUACIÓN DE PARCHES DE CÓDIGO GENERADOS POR LLMs MEDIANTE UN ENTORNO DE VERIFICACIÓN MULTICAPA

Autor: Ignacio Hernández Bas
Director: Furong Huang
Entidad Colaboradora: University of Maryland

RESUMEN DEL PROYECTO

Este trabajo propone un entorno de verificación multicapa para evaluar parches de código Python generados por LLMs mediante señales adicionales a la simple superación del conjunto de pruebas. El sistema combina: **análisis estático** acotado al parche, incluyendo un Índice de Calidad Estática (SQI) que agrega varias herramientas de análisis, **ejecución dinámica** en contenedores de las pruebas del repositorio, y una **capa semántica** que extrae afirmaciones de comportamiento a partir de la descripción de la incidencia y genera tests en pytest mediante un bucle de refinamiento guiado por un agente. Los tests se validan exigiendo que fallen en la versión con el bug y pasen en la implementación de referencia, denominada parche *gold*.

En SWE-bench Lite, el análisis estático se aplica a 300 parches de referencia, mientras que las capas dinámica y semántica se evalúan en 18 instancias con entornos disponibles. En conjunto, el 37,4% de los intentos produce tests discriminativos, la mejor combinación de modelos alcanza un 64,3% (18/28). Este resultado evidencia la utilidad de la capa semántica para detectar diferencias no cubiertas por los tests existentes, pero también este proyecto revela una limitación propia de la validación basada en referencia: los tests generados pueden quedar parcialmente anclados al comportamiento concreto del parche *gold*.

Palabras clave: Parches de Código Generados por LLMs, Verificación Multicapa, Evaluación de Corrección de Parches, Verificación Semántica, Pruebas Diferenciales, SWE-bench Lite

1. Introducción

Los recientes avances en modelos de lenguaje (LLMs) han mejorado significativamente la capacidad de generar parches de código para problemas reales de software [1]. Sin embargo, aunque benchmarks como SWE-bench [2] demuestran que estos modelos pueden producir soluciones que superan los tests existentes, dichas evaluaciones se limitan a la adecuación del conjunto de pruebas y no garantizan la correcta alineación con el comportamiento esperado descrito en las incidencias. Estudios recientes evidencian además que muchos problemas considerados como resueltos presentan errores semánticos [3].

Este trabajo aborda esta limitación mediante la propuesta de un marco de verificación que va más allá de la validación basada en tests. En lugar de depender exclusivamente de los tests existentes, el sistema introduce distintas capas de verificación que evalúan la corrección estructural, el com-

portamiento en ejecución y la coherencia con la especificación expresada en lenguaje natural. El objetivo es proporcionar una señal de evaluación más completa para los parches generados por LLMs.

2. Definición del Proyecto

El problema central abordado en este trabajo es la brecha existente entre la *corrección según tests* y la *corrección semántica*. Formalmente, dado un repositorio, una versión con error y un parche candidato, el objetivo es determinar si dicho parche satisface el comportamiento esperado descrito en la incidencia.

El proyecto se fundamenta en la siguiente hipótesis de verificación:

> Un pipeline de verificación multicapa que combine análisis estático, dinámico y semántico proporciona una evaluación más informativa y fiable de la corrección de parches que la validación basada únicamente en tests.

El sistema se evalúa utilizando SWE-bench Lite [2], que proporciona incidencias reales, parches asociados y entornos de ejecución. El análisis estático se aplica sobre el conjunto completo de datos, mientras que las capas dinámica y semántica se evalúan sobre un subconjunto de instancias debido a limitaciones de infraestructura.

3. Descripción del Sistema

El sistema propuesto se estructura como un entorno de verificación multicapa (véase Figura 3) compuesto por tres módulos secuenciales:

1. **Capa de Verificación Estática:** Realiza un análisis determinista del código sin ejecución. Incluye validación sintáctica, análisis estructural y un índice de calidad (SQI) basado en herramientas como Pylint, Radon, Flake8, Mypy y Bandit. Su objetivo es filtrar parches inválidos o de baja calidad.

2. **Capa de Verificación Dinámica:** Ejecuta los tests originales del repositorio en un entorno aislado. Verifica que el parche resuelve el error objetivo (*fail_to_pass*) sin introducir regresiones (*pass_to_pass*), siguiendo el protocolo estándar de SWE-bench [2].

3. **Capa de Verificación Semántica:** Introduce validación basada en el comportamiento descrito en la incidencia. Extrae afirmaciones estructuradas mediante un LLM, las vincula con el código y genera tests ejecutables mediante un proceso iterativo. Estos tests se validan mediante un esquema discriminativo (BUG falla, GOLD pasa), inspirado en técnicas de generación diferencial de tests [4,5]. Asimismo, trabajos recientes han demostrado la capacidad de los LLMs para generar tests directamente a partir de descripciones de incidencias [6].

El sistema sigue un proceso iterativo en el que la generación de tests se refina a partir del feedback obtenido durante la ejecución.

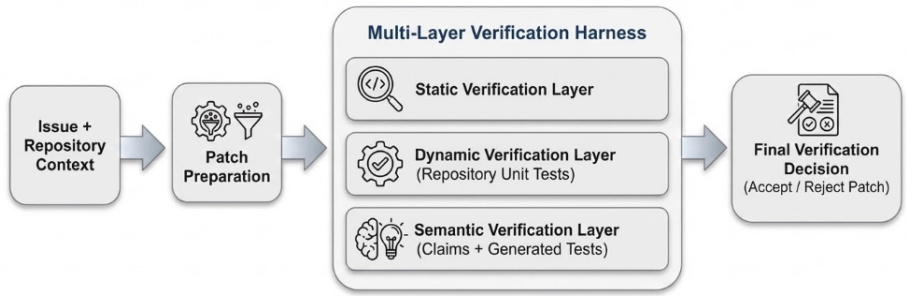

Figure 3: Esquema general del pipeline de verificación

4. Resultados

La capa de verificación estática se aplicó sobre las 300 instancias de SWE-bench Lite, proporcionando evaluaciones consistentes de calidad y estructura.

La capa dinámica se evaluó sobre un subconjunto de 18 instancias con entornos disponibles, alcanzando una alta tasa de éxito en la validación de parches correctos sin introducir regresiones.

La capa semántica demostró la capacidad de generar tests discriminativos que capturan el comportamiento esperado. Los resultados muestran que estos tests permiten detectar diferencias semánticas entre parches que superan los tests originales, una limitación conocida en la reparación automática de programas [5]. En particular, una proporción significativa de los parches candidatos alineados con la implementación de referencia fueron considerados consistentes con dicha implementación, mientras que otros fueron identificados como sobreajustados o divergentes.

Capa	**Instancias**	**Tasa de Éxito**	**Tipo de Señal**
Estática	300	100% válidas, 87,7% calidad alta	Calidad estructural
Dinámica	18	17/18 (94,4%) correctas	Validación mediante tests
Semántica	18	Hasta 64,3% discriminativa	Validación comportamental alineada con referencia

Table 2: Resumen de las capas de verificación

La Figura 4 muestra una comparación detallada de la tasa de discriminación para distintas combinaciones de modelos en las tareas de extracción de afirmaciones y generación de tests. Los resultados evidencian una fuerte dependencia del rendimiento respecto a la combinación utilizada, con variaciones significativas entre configuraciones.

La mejor combinación se obtiene utilizando Qwen3-32B tanto para la extracción de afirmaciones como para la generación de tests, alcanzando una tasa de discriminación del 64,3% (18/28). En contraste, otras combinaciones presentan rendimientos considerablemente inferiores, en algunos

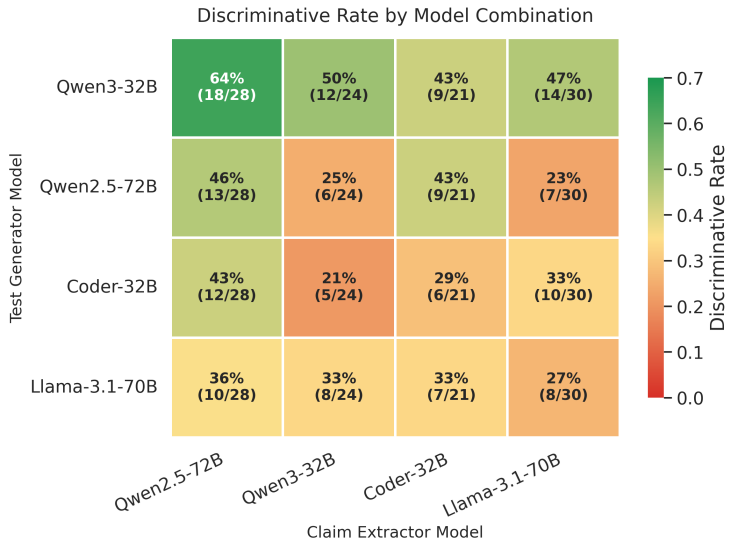

Figure 4: Tasa de discriminación según la combinación de modelos (generación de tests vs extracción de afirmaciones).

casos por debajo del 25%, lo que indica que ambas etapas del pipeline semántico contribuyen de forma crítica a la efectividad global del sistema.

Estos resultados sugieren que la calidad de la capa semántica no depende únicamente del modelo generador de tests, sino también de la estructura y relevancia de las afirmaciones extraídas. Una extracción de afirmaciones de mayor calidad permite generar tests más discriminativos y representativos del comportamiento esperado, mientras que afirmaciones más débiles limitan el rendimiento del sistema.

En conjunto, la variabilidad observada refuerza la importancia de optimizar conjuntamente ambas etapas de la capa semántica, ya que el rendimiento depende de forma crítica de la interacción entre la extracción de afirmaciones y la generación de tests.

Sin embargo, los resultados también revelan un fenómeno relevante: los tests generados tienden a alinearse con el comportamiento del parche de referencia. Este efecto, denominado *anclaje basado en la referencia*, indica que la validación es inherentemente relativa a la implementación de referencia y no a una especificación completa del problema.

Es importante destacar que superar la verificación dinámica no garantiza la corrección semántica, lo que refuerza la necesidad de validaciones adicionales.

En conjunto, el enfoque multicapa proporciona una señal de evaluación más rica que la validación basada únicamente en tests, permitiendo detectar diferencias comportamentales más sutiles.

5. Conclusiones

Este trabajo demuestra que la combinación de verificación estática, dinámica y semántica permite una evaluación más completa de los parches generados por LLMs. La capa semántica, en particular, amplía la validación tradicional al incorporar el comportamiento esperado descrito en lenguaje natural.

Los resultados confirman la hipótesis inicial: la verificación multicapa mejora la detección de errores no capturados por los tests existentes. No obstante, también se identifican limitaciones importantes, especialmente la dependencia de una implementación de referencia.

Además, este trabajo propone un marco modular y extensible para la verificación multicapa de parches, que puede servir de base para futuras investigaciones sobre la evaluación de sistemas basados en LLMs. Como líneas futuras, se propone explorar métodos de validación basados en especificaciones formales, múltiples implementaciones de referencia o evaluación humana, así como mejorar la escalabilidad y robustez del proceso de generación de tests.

6. Referencias

[1] Juyong Jiang et al. "A Survey on Large Language Models for Code Generation". In: *ACM Transactions on Software Engineering and Methodology* 35.2 (Jan. 2026), pp. 1–72. DOI: 10.1145/3747588.

[2] Carlos E. Jimenez et al. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" In: *The Twelfth International Conference on Learning Representations (ICLR)*. 2024. URL: https://openreview.net/forum?id=VTF8yNQM66.

[3] You Wang, Michael Pradel, and Zhongxin Liu. "Are "Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study". In: *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*. 2025. URL: https://arxiv.org/abs/2503.15223.

[4] Kunal Taneja and Tao Xie. "DiffGen: Automated Regression Unit-Test Generation". In: *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2008, pp. 407–410. DOI: 10.1109/ASE.2008.60.

[5] Qi Xin and Steven P. Reiss. "Identifying Test-Suite-Overfitted Patches through Test Case Generation". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2017, pp. 226–236. DOI: 10.1145/3092703.3092718.

[6] Sungmin Kang, Juyeon Yoon, and Shin Yoo. "Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction". In: *ICSE*. 2023. URL: https://arxiv.org/abs/2209.11515.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Software maintenance is one of the most resource-intensive activities in software engineering. A large proportion of developer effort is spent understanding, debugging, and repairing existing code rather than implementing new functionality. The economic impact of software defects is substantial: defects discovered in production can be an order of magnitude more expensive to fix than those detected during development [7]. Consequently, the automation of bug detection and repair has long been a central goal in software engineering research.

Recent advances in large language models (LLMs) have accelerated progress toward this goal. Models trained on large corpora of source code can now generate plausible patches for real-world bug reports and issues described in natural language. The SWE-bench benchmark [2] formalized this capability into a reproducible evaluation protocol: given a GitHub issue and the repository state before a human fix was committed, can a model generate a patch that passes the repository's test suite? As LLM-based agents have improved, they now resolve a non-trivial fraction of these instances, suggesting that LLM-assisted software repair is transitioning from an academic curiosity to an engineering reality.

However, the dominant evaluation protocol in SWE-bench, running the project's pre-existing test suite, carries a fundamental limitation: it conflates test passage with correctness. Passing the existing test suite does not necessarily imply that a patch correctly implements the intended behavioral change. Instead, it only confirms that the patch satisfies the specific scenarios captured by the existing tests. As a result, semantically incorrect patches may still be classified as correct if the test suite lacks sufficient behavioral coverage [3].

This observation exposes a *verification gap*. For a given bug fix, we ideally want tests that are specifically targeted at the behavioral claim addressed by the patch. More precisely, we want tests that are *discriminative*: they should fail on the buggy version of the code and pass on the patched version, thereby providing evidence that the patch performs the intended behavioral change.

Writing such targeted tests manually for every patch under evaluation is impractical at scale. A

natural question follows: can we automate this process using the same class of models that generated the patch? And if so, how do we ensure that the generated tests are trustworthy, neither trivially passing (non-discriminative) nor overly constrained by irrelevant implementation details?

To address this problem, this thesis proposes a multi-layer verification harness that integrates three complementary verification stages:

1. Static analysis of the generated patch.

2. Dynamic verification through repository unit tests.

3. Semantic verification based on behavioral claims extracted from issue descriptions.

These layers are orchestrated within a unified pipeline that evaluates candidate patches using progressively stronger verification signals.

1.1 Contributions

This work presents a multi-layer verification framework for evaluating LLM-generated code patches, combining static analysis, dynamic test execution, and semantic verification based on behavioral claims. The main contributions of this thesis are as follows:

- **Multi-layer verification harness.** We design and implement a unified verification pipeline that integrates static code analysis, dynamic execution of repository test suites, and semantic validation through LLM-generated claim-based tests. This architecture enables progressively stricter evaluation, moving from structural correctness to functional and behavioral verification.

- **Claim-based semantic verification.** We propose a method for extracting structured behavioral claims from natural-language issue descriptions and transforming them into executable test cases. These tests are validated through a discriminative execution scheme that compares behavior on buggy and patched versions of the code (as available in benchmark settings), providing a behavioral signal beyond traditional test-suite-based evaluation.

- **Agentic test-generation loop with discriminative validation.** We introduce an iterative, diagnosis-driven test-generation process that refines candidate tests based on execution feedback. The loop ensures that generated tests are discriminative (failing on the buggy version and passing on the patched version), enabling automatic validation of behavioral claims.

- **Empirical evaluation on SWE-bench Lite.** We evaluate the static layer on all 300 SWE-bench Lite instances and the dynamic/semantic layers on the 18 instances with pre-built execution images. Across multiple model configurations, claim-based tests provide a complementary signal to test-suite-based evaluation, achieving up to 64% discriminative success and highlighting cases where patches pass all tests but exhibit behavioral divergences.

- **Empirical characterization of reference-induced anchoring in LLM-driven test generation.** We show that the discriminative validation scheme (BUG=FAIL, GOLD=PASS) induces a selection pressure toward assertions aligned with the reference implementation. While related forms of reference-based validation have been studied in differential and regression testing, we provide an empirical analysis of this effect in the context of claim-driven, LLM-based semantic verification.

- **Reframing of semantic verification as reference-based validation.** Based on the observed anchoring effect, we show that claim-based verification should be interpreted as a form of *reference-based behavioral validation* rather than a fully specification-driven notion of correctness. This insight highlights a fundamental limitation of current evaluation paradigms in automated program repair and suggests directions for future work in specification-grounded and multi-reference verification.

Together, these components aim to strengthen the verification of LLM-generated patches by combining structural analysis, behavioral testing, and semantic reasoning within a unified evaluation pipeline.

1.2 Thesis Structure

The remainder of this thesis is organized as follows:

- **Chapter 2 (State of the Art)** reviews the literature on automated program repair, LLM-based code generation, automated test generation, and agentic AI systems for software engineering.

- **Chapter 3 (Project Definition)** presents the motivation, objectives, and methodological framework of the project, defining the verification problem addressed in this work.

- **Chapter 4 (Technology Stack & Resources)** describes the technologies, resources, and tools used in the implementation of the pipeline.

- **Chapter 5 (System Design & Architecture)** describes the design and implementation of the proposed verification pipeline, including claim extraction, multi-layer verification, and the agentic feedback loop.

- **Chapter 6 (Results Analysis)** presents the empirical evaluation and analyzes the effectiveness of the proposed approach.

- **Chapter 7 (Discussion and Future Work)** discusses the main findings, limitations of the current system, and possible directions for future research.

- **Chapter 8 (Conclusions)** summarises the contributions and key outcomes of the thesis.

Chapter 2

State of the Art

This chapter reviews the research areas most relevant to this work: automated program repair, LLM-based code generation and patching, automated test generation, specification extraction from natural language, and agentic AI systems for software engineering. We conclude by positioning our contribution relative to the existing literature.

2.1 Automated Program Repair

Automated program repair (APR) aims to automatically generate modifications to a software artifact that cause it to satisfy a correctness criterion, typically a test suite. Early APR approaches relied on genetic programming, search-based techniques, or template-based strategies, exploiting common edit patterns observed in human-written patches (e.g., GenProg, PAR) [8].

A key limitation of test-suite-driven APR is *overfitting*: a generated patch may pass all provided tests while being semantically incorrect, for instance by deleting code under test or special-casing test inputs [8]. This "patch correctness" problem is directly related to this thesis. While our system generates *tests* rather than patches, it faces a symmetric risk: a generated test may be formally valid (e.g., it distinguishes bug from fix) while remaining semantically fragile (e.g., it overfits to implementation details rather than the behavioral intent).

To mitigate overfitting, semantic APR approaches incorporate additional constraints such as formal specifications, invariants, or information from bug reports [8]. In modern LLM-based settings, natural-language issue descriptions provide an additional and largely orthogonal source of semantic constraint, which motivates treating issue text as a weak form of specification.

2.2 LLM-Based Code Generation and Patch Synthesis

Large language models trained on code demonstrated strong capabilities for program synthesis and code completion [9]. Subsequent work has surveyed the rapid development of LLM-based code generation systems and their applications in software engineering [1]. These models have also been applied to automated program repair by leveraging commit diffs and natural-language issue descriptions as supervision signals.

Patch quality has been formalized at scale through SWE-bench, which pairs real-world GitHub issues with the corresponding human fix and evaluates candidate patches by running the repository's existing test suite [2]. SWE-bench-Lite and SWE-bench-Verified provide more tractable subsets and partially strengthen evaluation [10], but the core protocol remains test-centric.

Recent empirical studies have shown that even patches considered solved in SWE-bench may not always be semantically correct, highlighting the difficulty of evaluating patch correctness solely through existing tests [3].

Agentic repair systems improve patch generation by incorporating repository navigation, localization, and multi-step planning. Autonomous repair agents capable of iteratively exploring repositories and generating patches have recently been proposed [11]. Nevertheless, these systems are still largely evaluated using pre-existing tests, and thus inherit the limitation that test passage can overestimate semantic correctness when behavioral intent is not fully captured by the test suite.

2.3 Automated Test Generation

Automated test generation has a long history in software engineering. Coverage-driven approaches (e.g., random testing and evolutionary testing) optimize for structural coverage metrics such as branch or statement coverage. While effective for exercising code, they are not designed to target the behavioral intent of a specific change.

Mutation testing evaluates test quality based on the ability to distinguish a program from mutants (small syntactic modifications) [12]. The discriminative test generation problem studied in this thesis can be viewed as a special case where the "mutant" corresponds to the pre-patch (buggy) version and the reference corresponds to the post-patch (fixed) version, except that the behavioral change is a real defect fix rather than an artificial perturbation.

Regression and differential test generation explicitly target behavioral differences between program versions. DiffGen instruments two versions of a class and generates unit tests that expose behavioral differences between them [4]. Similarly, regression testing techniques implemented in tools such as EvoSuite generate assertions that encode the behavior of a reference version, enabling tests that are expected to pass on the reference and fail on alternative or modified versions [13]. These approaches illustrate that reference-based test validation is a long-standing strategy in software testing.

In particular, these methods share a fundamental principle with the semantic verification layer proposed in this work: correctness is evaluated relative to a reference implementation. In both differential testing and overfitting patch detection, a test is considered meaningful if it exposes a behavioral difference between two program versions.

The semantic layer can therefore be understood as a form of *reference-based differential validation*, in which correctness is defined relative to a reference implementation and tests are required to distinguish between program behaviors. Unlike prior approaches, which typically rely on coverage objectives or syntactic transformations, the proposed method derives tests from natural-language behavioral claims and validates them under a BUG-versus-GOLD execution scheme. This connection clarifies that the proposed approach extends existing paradigms by integrating natural-language grounding and LLM-driven synthesis into a reference-relative validation framework.

Recent work applies LLMs to unit test generation and verification. For example, ChatUniTest automatically generates unit tests from method specifications using large language models [14]. Other systems aim to improve existing test suites using coverage feedback and iterative test synthesis, such as CoverAgent [15]. Some approaches also use generated tests to validate or rerank model outputs, combining test synthesis with program verification [16, 17]. However, generating tests that are both *discriminative* across versions and aligned with the intended behavioral change remains less studied in comparison to coverage-oriented or docstring-driven unit test generation.

2.4 Specification Extraction and Behavioral Claims

A persistent challenge in test generation is the absence of machine-readable specifications. Property-based testing requires developers to explicitly write properties and generators, while specification mining attempts to infer likely invariants automatically from program executions [18]. These inferred invariants may capture useful behavioral properties but are often execution-dependent and may fail to represent rare behaviors.

Recent work explores using LLMs to derive constraints or specifications from natural-language documentation and to guide fuzzing or contract generation. However, the use of natural language, such as issue descriptions or commit messages, as sources of structured behavioral specifications remains relatively underexplored. These artifacts frequently describe the intended behavioral change of a patch in natural language, yet they are rarely transformed into explicit, machine-actionable claims that can guide automated verification or test generation.

Extracting behavioral claims from issue descriptions and grounding them to concrete program entities (e.g., functions, classes, or methods) may provide a bridge between human-readable descriptions of intended behavior and executable tests. Such grounded claims could serve as weak specifications that help guide the generation of tests aligned with the behavioral intent of a change.

2.5 Agentic AI Systems for Software Engineering

Recent research has proposed autonomous agents capable of navigating repositories, executing commands, and iteratively repairing code. Systems such as SWE-agent enable LLMs to interact with development environments, run commands, and generate patches in order to resolve real-world GitHub issues [19]. Other approaches attempt to simplify this interaction paradigm, for example, Agentless, which reduces the reliance on complex tool-use loops [20]. Similarly, generate-and-debug frameworks such as Self-Debug demonstrate how LLMs can iteratively improve generated code by analyzing execution feedback [21]. Autonomous repair agents capable of iteratively exploring repositories and generating patches have also been proposed in the automated program repair literature [11].

A common design pattern in these systems is a closed-loop interaction between generation and feedback: a model proposes an action (e.g., a patch or code modification), execution feedback is collected, and the model iteratively refines its output based on the observed outcome. While such agentic loops have primarily been applied to patch generation and debugging, their use for guiding automated test synthesis and verification remains comparatively underexplored.

2.6 Evaluation of Patch Correctness

Several approaches attempt to improve the verification of automatically generated patches by introducing additional validation mechanisms. For instance, LIBRO proposes using LLM-generated tests to verify candidate patches against behavioral expectations [6].

A closely related line of work focuses on detecting test-suite-overfitted patches using reference patches as oracles. DiffTGen identifies overfitted patches by generating tests that distinguish a machine patch from the human-written fix [5]. At larger scale, automated patch assessment based on random testing with ground truth (RGT) uses the human patch to generate tests and flags candidate patches that diverge from the reference behavior [22]. These methods mirror the reference-relative validation scheme used in the semantic layer of this thesis.

In this setting, the human-written patch effectively acts as a behavioral oracle, defining the expected outcome against which alternative implementations are evaluated.

Beyond SWE-bench, prior work has proposed evaluating patch correctness using additional tests, hidden test cases, or differential testing methodologies. Differential testing approaches evaluate patches by comparing program behavior across versions [3]. Security-focused evaluation frameworks have also proposed specialized benchmarks to analyze vulnerabilities and verify security patches generated by LLMs [23].

An emerging research question is whether tests can be generated that explicitly target the behavioral intent of a change rather than merely increasing structural coverage. Ideally, such tests would discriminate between program versions by analyzing behavioral claims expressed in issue descriptions or commit context. However, systematic methods for generating intent-aligned discrimina-

tive tests remain limited, highlighting an important opportunity for improving the evaluation of automatically generated patches.

2.6.1 Multi-Layer Verification Pipelines

Several recent systems have explored combining multiple verification signals when evaluating automatically generated code. Evaluation infrastructures for LLM-generated programs often integrate static analysis, execution-based testing, and additional verification mechanisms in order to improve reliability [24].

Static analysis tools provide lightweight checks that detect syntactic errors, type inconsistencies, and potential security vulnerabilities without executing the program. Dynamic testing frameworks complement these techniques by executing unit tests or fuzzing inputs to evaluate program behavior. However, both approaches remain limited when the intended behavioral change is not fully captured by existing tests.

This observation motivates layered verification architectures in which progressively stronger analysis techniques are applied to candidate patches. Early layers provide fast structural validation, while later layers perform deeper behavioral verification. Such layered verification pipelines combine static analysis, dynamic testing, and additional semantic verification signals in order to improve the reliability of automated patch evaluation.

2.7 Summary and Positioning

Table 2.7.1 highlights that existing approaches either focus on test generation, reference-based validation, or agentic refinement, but do not combine these dimensions. The proposed system integrates all of these aspects into a unified framework for claim-driven discriminative verification.

Table 2.7.1: Positioning of this work relative to closely related systems.

System	Task	Test gen.	Reference-based	Discriminative	Claim-grounded	Closed-loop	Multi-model
SWE-agent [19]	Patch generation	–	–	–	–	✓	–
Agentless [20]	Patch generation	–	–	–	–	–	–
ChatUniTest [14]	Unit test generation	✓	–	–	–	–	–
LIBRO [6]	Patch validation	✓	✓	–	–	–	–
CoverAgent [15]	Test improvement	✓	–	–	–	✓	–
Self-Debug [21]	Code repair	–	–	–	–	✓	–
DiffTGen [5]	Patch validation	✓	✓	✓	–	–	–
RGT [22]	Patch validation	✓	✓	✓	–	–	–
This work	**Discriminative test gen.**	✓	✓	✓	✓	✓	✓

The key differentiators of this work are: (1) an explicit focus on discriminative tests anchored to a natural-language behavioral claim, (2) a multi-layer verification architecture that combines static guardrails with execution-driven feedback, and (3) a systematic evaluation that decouples the claim extraction and test synthesis roles.

Chapter 3

Project Definition

3.1 Motivation

Despite significant progress in the evaluation of AI-generated code, current verification approaches remain fundamentally limited in their ability to assess semantic correctness. Test-based evaluation assumes that the existing test suite fully captures the intended behavior of the system, an assumption that often does not hold in practice. Static analysis techniques improve code quality and safety, but do not guarantee behavioral alignment with user expectations. Differential testing frameworks offer stronger guarantees but depend on the availability of oracle patches, which are rarely accessible in real-world deployment scenarios.

A key observation motivating this work is that software issues are primarily specified in natural language. Issue descriptions frequently contain explicit or implicit statements about expected behavior, constraints, and edge cases that are not exhaustively encoded in executable tests. As a result, the semantic intent of the user is often present in textual form but remains unused during verification. Ignoring this information leads to situations in which patches that technically pass all tests nonetheless violate the original requirements of the issue, a limitation that persists even in the previously mentioned benchmarks augmented with human-validated tests [3, 10].

At the same time, prior attempts to infer semantic correctness solely from program executions, execution traces, or inferred invariants face practical challenges. Such approaches may struggle to distinguish between correct and incorrect implementations for logical bugs, particularly when execution coverage is limited or when inferred properties are too weak to characterize the intended behavior. This observation is closely related to prior work on specification mining, where inferred properties are treated as weak specifications derived from partial evidence rather than formal correctness guarantees [18]. More generally, Rice's theorem implies that no general procedure can decide non-trivial semantic properties of programs, reinforcing the need for approximate and probabilistic verification strategies [25].

Recent advances in Natural Language Processing provide an opportunity to bridge this gap. Modern NLP models for source code and natural language jointly learn representations that enable structured information to be extracted from informal textual descriptions [26, 27]. These advances make it feasible to treat issue descriptions as weak specifications and to extract behavioral statements that can be treated as verification targets.

This project is motivated by the hypothesis that incorporating behavioral information derived from issue descriptions can improve the verification of automatically generated patches. Rather than relying solely on test-suite execution or low-level program analysis, the proposed approach extracts structured behavioral claims and translates them into executable tests.

However, instead of providing a fully specification-driven notion of correctness, the approach approximates behavioral validation through discriminative testing against a reference implementation. In this setting, a patch is considered correct if it satisfies behavioral claims that distinguish it from the buggy version and align with the expected behavior observed in the reference solution.

This perspective reframes verification as a comparison between behaviorally grounded executions, rather than as a complete formal validation of user intent.

By integrating claim extraction and grounding into a layered verification harness, the approach aims to complement existing static and dynamic checks with a semantic layer driven directly by natural-language intent. This design seeks to reduce false acceptances and provide a more faithful assessment of whether an AI-generated patch truly resolves the issue it claims to address.

3.2 Problem Formulation

This thesis addresses the following core problem:

> Given a buggy repository state, its issue description, and a reference (gold) fix available in benchmark settings, automatically generate a *discriminative behavioral test* that fails on the pre-patch version and passes on the reference-fixed version, thereby verifying that the patch implements the intended behavioral change under the reference oracle.

We decompose this problem into three sub-problems:

1. **Claim extraction:** From a commit diff and associated metadata (issue title, issue body, changed files), extract a precise natural-language claim about the behavioral change introduced by the patch. This claim serves as a weak specification that the test must verify.

2. **Test synthesis:** Given a claim, the repository context, and a set of structural constraints, generate a *pytest* test that converts the claim into an executable assertion.

3. **Discriminability verification:** Execute the generated test against both the buggy (pre-

patch) and fixed (reference) versions of the repository. Classify the result and, if the test is not discriminative, generate targeted feedback to guide the next synthesis attempt.

The third sub-problem introduces a closed-loop structure: test synthesis and verification alternate until a discriminative test is found or a maximum number of attempts is exhausted.

In this work, discriminative validation relies on executing tests against both the buggy and reference-fixed versions. This assumption aligns with benchmark datasets such as SWE-bench, but it also bounds the scope of the semantic layer to settings where a reference implementation is available or where human validation can stand in as the oracle.

3.2.1 Research Questions

To make the scope explicit, we frame the investigation around three research questions:

- **RQ1:** Can natural-language issue descriptions be converted into grounded behavioral claims that are specific enough to guide test generation?
- **RQ2:** Can an agentic loop reliably generate discriminative tests that fail on the buggy version and pass on the reference-fixed version?
- **RQ3:** Do discriminative claim-based tests detect behavioral divergences among candidate patches that pass the existing test suite?

3.3 Objectives

The primary objective of this project is to design and implement a multi-layer verification harness capable of evaluating LLM-generated code patches beyond traditional test-based validation.

The specific objectives of the project are:

- Design a modular verification architecture supporting multiple analysis layers.
- Implement a dataset ingestion pipeline compatible with SWE-bench Lite and related benchmarks.
- Develop a static verification layer to analyze code structure, quality, and potential defects.
- Develop an NLP-based pipeline to extract behavioral claims from natural-language issue descriptions.
- Define a grounding mechanism linking extracted claims to the code elements modified by a patch.

- Generate executable tests from grounded claims and integrate them into the dynamic verification process.
- Evaluate the effectiveness of claim-based verification in reducing false acceptances relative to test-only evaluation.
- Assess the feasibility, limitations, and practical impact of natural-language-driven semantic verification.

These objectives balance theoretical soundness with practical applicability. The experimental evaluation focuses on benchmark settings where a reference (gold) patch is available; in deployment, the static and dynamic layers remain applicable, while the semantic layer requires either a reference implementation or additional human validation.

Chapter 4

Technology Stack & Resources

This chapter describes the technological resources used to implement the proposed verification harness. The system integrates several components spanning datasets, software analysis tools, large language models, and execution infrastructure. Together, these elements enable the construction of a scalable pipeline capable of evaluating AI-generated patches through static analysis, dynamic testing, and semantic verification.

4.1 Dataset

The experimental evaluation in this thesis is based on the **SWE-bench Lite** benchmark. SWE-bench is a large-scale dataset designed to evaluate the ability of language models to resolve real-world GitHub issues by generating code patches.

Each instance in SWE-bench corresponds to a historical bug fix in a public repository and includes:

- The GitHub issue describing the problem.
- The repository state prior to the fix.
- The human-written patch that resolves the issue.
- A command for executing the repository's test suite.

SWE-bench Lite provides a curated subset of the full dataset containing several hundred instances across multiple widely used Python repositories. These instances are sufficiently complex to represent realistic software maintenance tasks while remaining computationally tractable for experimental evaluation.

The dataset is accessed through the **HuggingFace Hub** using the **datasets** library, which provides

efficient loading and caching mechanisms for large datasets.

4.2 Software Repositories

The dataset instances originate from several widely used open-source Python projects. These repositories span different application domains, including web frameworks, scientific computing libraries, documentation systems, and developer tooling.

Examples of repositories included in the dataset include: Django [28], Matplotlib [29], and Scikit-learn [30].

For each evaluation instance, the repository is checked out at the specific commit corresponding to the state before the bug fix was introduced. This ensures that candidate patches are evaluated in a reproducible environment consistent with the original issue.

4.3 Large Language Models

Large language models (LLMs) play a central role in the semantic verification layer of the proposed system. In particular, they are used to extract behavioral claims from issue descriptions and to generate executable tests that verify those claims.

Several open-weight instruction-tuned models specialized in code understanding and generation are used in the experiments. These include models from the Qwen and LLaMA families. Using multiple models enables the evaluation of different model capabilities for claim extraction and test generation.

Model inference is served through the **vLLM** [31] framework, which provides efficient GPU-based inference and exposes an API compatible with the OpenAI interface. This design allows the system to easily switch between different models while maintaining a consistent interaction interface.

To reduce memory requirements and improve inference efficiency, the models are loaded using quantized weights. Quantization techniques reduce the precision of model parameters while preserving most of the model's predictive performance, enabling large models to run on a limited number of GPUs.

4.4 Static Analysis Tools

The static verification layer relies on several established static analysis tools for Python. These tools evaluate code quality, detect potential defects, and measure structural properties of the modified code without executing the program.

The main tools used in this layer include:

- **Pylint**[32], which performs comprehensive static analysis and detects logical errors and potential programming mistakes.
- **Flake8**[33], which checks compliance with Python style guidelines and identifies formatting and syntactic issues.
- **Radon** [34], which measures code complexity and maintainability using metrics such as cyclomatic complexity.
- **MyPy** [35], which performs static type checking in projects that include type annotations.
- **Bandit** [36], which scans Python code for common security vulnerabilities.

Each tool captures different aspects of software quality. Their outputs are combined into a composite metric called the *Static Quality Index* (SQI), which provides an aggregated indicator of patch quality.

4.5 Dynamic Testing Framework

Dynamic verification evaluates candidate patches by executing unit tests in an isolated environment. This stage mirrors the evaluation procedure used in SWE-bench-style benchmarks, where patches are validated by running the repository's existing test suite.

For most repositories in the dataset, test discovery and execution are performed using the **pytest** [37] testing framework. Pytest is widely adopted in Python projects and provides flexible mechanisms for test discovery, execution, and reporting.

However, some repositories rely on project-specific testing infrastructures. For example, the **Django** project uses its own test runner built on top of Python's **unittest** framework, while certain projects provide custom wrappers for executing tests. In these cases, the verification pipeline invokes the repository's native test command to reproduce the original evaluation procedure defined by the benchmark.

4.6 Execution Infrastructure

Experiments are executed on a GPU-enabled computing cluster that supports large-scale evaluation of candidate patches. Job scheduling and resource allocation are managed using the **SLURM** [38] workload manager, which enables parallel execution across multiple compute nodes.

To ensure reproducibility and isolation of execution environments, the system relies on containerization technologies. Containers encapsulate the required dependencies, libraries, and runtime configurations necessary to execute repository tests and generated code.

For high-performance computing environments, containers are executed using **Singularity** [39], which is widely adopted in research clusters due to its compatibility with multi-user systems and its ability to run without requiring root privileges.

This infrastructure enables the evaluation pipeline to scale across many instances while maintaining reproducibility and environment consistency.

4.7 Visualization and User Interface

To facilitate the exploration of experimental results, the system includes an interactive visualization interface implemented using **Streamlit** [40]. Streamlit enables the rapid development of web-based dashboards directly in Python without requiring a separate frontend framework.

The dashboard allows users to inspect verification results, compare model performance, and analyze the outcomes of the different verification layers. Visualization libraries such as Plotly [41] and Altair [42] are used to generate interactive charts and statistical summaries.

This interface provides a convenient way to analyze the behavior of the verification harness and to interpret the results obtained during experimental evaluation.

4.8 Summary

The technology stack described in this chapter provides the foundation for implementing the multi-layer verification harness proposed in this thesis. The system combines a realistic software engineering benchmark, advanced language models, established static and dynamic analysis tools, and scalable computing infrastructure.

The following chapter describes how these components are integrated into a unified architecture that performs patch verification through static analysis, dynamic execution, and semantic evaluation.

Chapter 5

System Design & Architecture

This chapter describes the architecture of the proposed multi-layer verification harness. The system combines static analysis, dynamic execution, and semantic verification based on behavioral claims in order to evaluate the correctness of LLM-generated code patches.

5.1 Overview of the Verification Pipeline

The proposed system follows a multi-layer verification approach in which candidate patches are evaluated. Figure 5.1.1 illustrates the architecture of the proposed harness. First, LLM-generated patches and repository context are obtained from the SWE-bench benchmark. After patch preparation and preprocessing, the candidate patch is analyzed sequentially by three verification layers: static verification, dynamic verification, and semantic verification.

Each layer performs different checks and validation procedures designed to assess whether the candidate patch correctly resolves the issue described in the repository. The results of these verification stages are aggregated to produce the final verification decision, determining whether the patch is accepted or rejected.

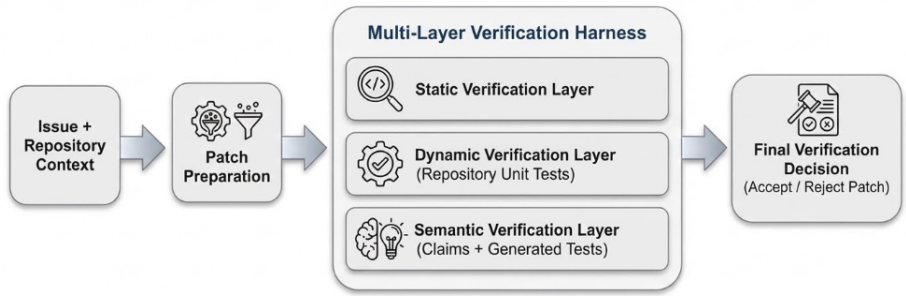

Figure 5.1.1: Verification Pipeline Overview

5.2 Context and Patch Preparation

Before any verification layer can be applied, the system must assemble the input artifacts that each layer consumes. This section describes the structure of a SWE-bench instance and the preprocessing steps that transform raw benchmark data into a form suitable for multi-layer verification.

5.2.1 SWE-bench Instance Structure

The most relevant fields that a SWE-bench Lite instance contains are the following:

- **instance_id**: A unique identifier (e.g., *astropy__astropy-6938*).
- **repo**: The GitHub repository path (e.g., *astropy/astropy*).
- **base_commit**: The commit hash of the repository state *before* the bug fix was applied. This defines the buggy (pre-patch) version.
- **patch**: The human-written fix in unified diff format, specifying the exact lines added, removed, or modified.
- **problem_statement**: The natural-language issue description from the GitHub issue tracker, which contains the user's report of the bug and, in many cases, implicit or explicit behavioral expectations.
- **test_patch**: A set of unit tests added by the repository developers alongside the fix, used by the benchmark as an oracle for evaluation.
- **FAIL_TO_PASS**: A list of fully qualified test identifiers (e.g., *module/tests/test_foo.py::test_bar*) that *fail* on the buggy (pre-patch) version and *pass* after the fix is applied. These are the tests that the human-written patch was designed to fix and serve as the primary oracle in the SWE-bench evaluation protocol.
- **PASS_TO_PASS**: A list of test identifiers that *pass* on both the pre-patch and post-patch versions. These regression tests ensure that the fix does not break existing functionality. A valid patch must keep all of these tests passing.
- **environment_setup_commit**: The commit used to configure the execution environment (dependencies, build scripts) for reproducing and testing the instance.

The *patch* and *problem_statement* fields are the primary inputs to the verification harness. The patch defines what changed, while the problem statement describes why it changed.

5.2.2 Repository Checkout and Code Context Extraction

To enable verification layers that require access to the source code (rather than just the diff), the preparation stage reconstructs the pre-patch repository state. For each instance, the system clones the corresponding repository and checks out the *base_commit*, producing a local copy of the codebase as it existed before the fix.

From this checkout, the system extracts two forms of code context:

1. **Windowed code context.** The unified diff is parsed to identify the files and line ranges modified by the patch. For each modified region, a configurable window of surrounding lines (30 lines before and after each hunk by default) is extracted from the pre-patch source. Adjacent or overlapping windows are merged to avoid redundancy. This windowed context provides the local code neighborhood around each change, which is used by the static layer for structural analysis and by the semantic layer as grounding context for claim extraction.

2. **Full touched-file text.** The complete source of each file modified by the patch is also loaded. This is necessary for static analysis tools (e.g., AST parsing, linting) that require syntactically complete files in order to produce meaningful results.

Both forms of context are subject to configurable size limits to prevent excessively large instances from consuming disproportionate memory or exceeding the context windows of downstream language models.

5.2.3 Enriched Instance Representation

The output of the preparation stage is an enriched instance that augments the original SWE-bench fields with two additional artifacts:

- *code_context*: The windowed source excerpts surrounding each diff hunk, formatted with file paths and line numbers.

- *touched_files_text*: A mapping from relative file paths to the full pre-patch source text of each modified file.

This enriched representation is serialized as JSON and serves as the unified input to all three verification layers. The static layer consumes the full file texts and the diff; the dynamic layer applies the patch to the checked-out repository and executes tests; and the semantic layer uses the problem statement, diff, and windowed code context to extract and ground behavioral claims.

Figure 5.1.1 shows where this preparation stage sits relative to the verification layers.

5.3 Static Verification Layer

The static verification layer constitutes the first analytical stage of the proposed multi-layer verification harness. This layer evaluates candidate patches using static code analysis techniques that operate directly on the source code without executing the program. The goal of this stage is to quickly detect structurally invalid or low-quality patches before engaging more computationally expensive verification procedures such as dynamic testing or semantic analysis.

Static analysis is particularly well-suited for this role because it is deterministic, computationally inexpensive, and capable of identifying a broad class of structural and quality issues in source code. By filtering out syntactically incorrect or structurally problematic patches early in the pipeline, the system reduces unnecessary computation in later verification stages and improves overall evaluation efficiency.

The static layer performs three main tasks:

- Patch Scoping
- Syntactic and Structural analysis
- Code Quality assessment

Implementation details of the static analysis pipeline, including the diff parsing procedure, AST-based structural analysis, and the full computation of the Static Quality Index, are provided in Appendix A.

5.3.1 Patch Scoping and Input Processing

Candidate patches are provided as unified diffs (see Listing 5.1), which specify the lines of code modified by the patch relative to the repository version associated with the issue.

The first step of the static layer is to identify the files affected by the patch and determine the specific regions of code that have been modified. This information allows the verification process to focus exclusively on the parts of the codebase that were actually changed by the candidate patch.

Because the evaluation dataset consists of Python repositories, the analysis is restricted to Python files. This decision avoids unnecessary processing of auxiliary files such as configuration files or documentation that may appear in the diff but are irrelevant to the functional correctness of the patch.

```
1 diff --git a/astropy/wcs/wcs.py b/astropy/wcs/wcs.py
2 --- a/astropy/wcs/wcs.py
3 +++ b/astropy/wcs/wcs.py
4 @@ -1212,6 +1212,9 @@ def _array_converter(self, func, sky, *args,
      ra_dec_order=False):
5 """
```

```
6
7  def _return_list_of_arrays(axes, origin):
8  + if any([x.size == 0 for x in axes]):
9  + return axes
10 +
11 try:
12 axes = np.broadcast_arrays(*axes)
13 except ValueError:
14 @@ -1235,6 +1238,8 @@ def _return_single_array(xy, origin):
15 raise ValueError(
16 "When providing two arguments, the array must be "
17 "of shape (N, {0})".format(self.naxis))
18 + if 0 in xy.shape:
19 + return xy
20 if ra_dec_order and sky == 'input':
21 xy = self._denormalize_sky(xy)
22 result = func(xy, origin)
```

Listing 5.1: Diff example

5.3.2 Syntax Validation

The most fundamental validation performed by the static layer is syntax checking. Each modified file is parsed using Python's Abstract Syntax Tree (AST) infrastructure.

If a file cannot be successfully parsed, the patch is immediately marked as syntactically invalid. This early rejection mechanism prevents malformed patches from propagating to later verification stages, where execution would fail.

Using AST parsing provides two advantages. First, it offers a fast and reliable mechanism for detecting syntax errors without executing the code. Second, it produces a structured representation of the program that can be reused to extract additional structural metrics about the modified code.

5.3.3 Structural Analysis

For patches that pass syntax validation, the AST representation of the modified files is analyzed to extract structural characteristics of the code.

The analysis focuses on high-level indicators that describe the organization and complexity of the code, including:

- Number of functions and classes defined in the file
- Nesting depth of the program structure
- Average length of functions

These metrics provide a lightweight approximation of code complexity and maintainability. Although they are not used as strict rejection criteria, they are recorded as diagnostic signals in the patch report and passed forward as contextual evidence for the subsequent verification layers.

In addition to these metrics, the system identifies which functions are affected by the patch by comparing the modified line ranges with the boundaries of function definitions in the AST.

To quantify how extensively a patch modifies the structure of a file, the system computes a structural change ratio based on the functions affected by the patch.

This ratio measures the proportion of functions or classes that overlap with the modified lines relative to the total number of definitions in the file:

$$\text{AST Diff Ratio} = \frac{\text{Number of modified functions}}{\text{Total number of functions and classes}}$$

Values close to zero indicate localized modifications affecting only a small portion of the code, whereas higher values indicate broader structural changes.

The metric is not used to directly accept or reject patches. Instead, it provides contextual information about the potential impact and risk associated with the modification.

5.3.4 Code Quality Analysis

The overall quality of the modified code is evaluated using several established static analysis tools introduced in section 4.4. These tools provide complementary signals that capture different dimensions of software quality, including logical correctness, style compliance, code complexity, type consistency, and potential security vulnerabilities.

In particular, the combination of analyzers enables the system to evaluate multiple aspects of code quality simultaneously. For example, tools focused on semantic correctness and potential programming errors complement those enforcing coding standards and maintainability metrics, providing a broader view of patch quality than any single analyzer could offer.

Static Quality Index (SQI)

Because each analyzer produces outputs with different scales and formats, their results are normalized and combined into a single composite metric called the **Static Quality Index (SQI)**.

The SQI aggregates normalized signals from the different analyzers through a weighted combination that reflects the relative informativeness of each signal. Tools that capture broader aspects of code correctness and maintainability receive higher weights, while tools that produce weaker or more context-dependent signals contribute smaller weights. These weights can be adjusted depending on the requirements of the evaluation scenario.

The resulting score is defined in the range [0,100] where higher values correspond to higher code quality. To facilitate interpretation, the SQI is mapped to qualitative categories:

SQI Range	Quality Level
85 – 100	Excellent
70 – 85	Good
50 – 70	Fair
< 50	Poor

Table 5.3.1: Static Quality Index (SQI) classification

5.3.5 Role of the Static Layer

The static verification layer acts as a **fail-fast filter** within the multi-layer architecture of the verification harness. Its primary function is to eliminate syntactically invalid or structurally problematic patches before more expensive verification procedures are executed. Patches whose Static Quality Index falls within the *Poor* category (SQI < 50) are rejected at this stage and are not forwarded to the subsequent verification layers.

Because static analysis requires no program execution, it can be applied efficiently to large numbers of candidate patches. This makes it particularly valuable when evaluating many patches across multiple repositories, where early rejection significantly reduces the computational cost of the overall pipeline.

Furthermore, the diagnostic information produced by the static layer, such as complexity metrics and quality scores, provides additional context that can assist both automated and human analysis of patch quality

5.4 Dynamic Verification Layer

Following static analysis, candidate patches are evaluated through dynamic verification by executing the repository's unit tests in an isolated environment. This stage follows the evaluation procedure adopted by SWE-bench-style benchmarks and serves as the primary functional correctness check. Figure 5.4.2 illustrates the workflow of the dynamic verification layer.

As described in Section 5.2.1, SWE-bench provides two categories of tests:

- **FAIL_TO_PASS**: tests that fail on the buggy version and should pass after applying the correct fix.
- **PASS_TO_PASS**: tests that pass before the patch and must remain successful afterwards.

A patch passes dynamic verification only if all tests in both categories produce their expected outcomes. Patches that fail this stage are rejected and do not proceed to semantic verification.

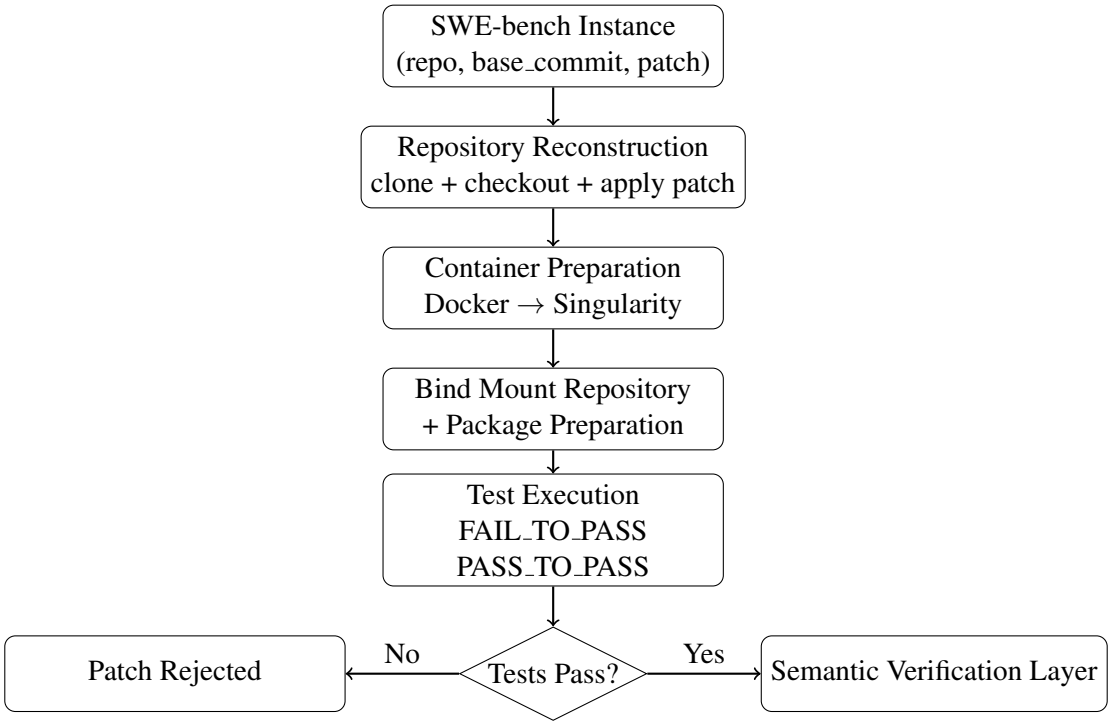

Figure 5.4.2: Dynamic verification pipeline for evaluating candidate patches.

Although necessary, test-suite validation alone is insufficient to guarantee semantic correctness. Repository developers wrote existing tests to validate their intended fix and may not fully capture the behavioral intent of the issue. This limitation motivates the additional semantic verification layer introduced later in the pipeline.

5.4.1 Repository Cloning and Patch Application

The first step in dynamic verification is reconstructing the repository state against which the patch must be evaluated. Each SWE-bench instance specifies a repository and a *base_commit* representing the version of the codebase at which the issue was reported.

The system performs the following steps:

1. Clone the target repository.

2. Checkout the specified base commit.

3. Apply the candidate patch generated by the model.

4. Apply any additional test patches provided by SWE-bench.

To enable large-scale evaluation on shared infrastructure, each instance is processed in an isolated

working directory. This prevents filesystem conflicts when multiple evaluations run concurrently.

5.4.2 Containerized Execution Environment

Reproducing the exact software environment required by each repository is a significant challenge. SWE-bench addresses this problem by providing pre-built Docker images containing fully configured environments with all required dependencies.

Because the evaluation is performed on a High-Performance Computing (HPC) cluster, these Docker images are converted to the Singularity Image Format (*.sif*). Singularity is designed for HPC environments, and allows containers to run as unprivileged user processes without requiring a daemon or root privileges.

Using Singularity provides several advantages over Docker in an HPC context:

- **Security**: Docker requires a privileged daemon with root access to the host kernel, which is incompatible with multi-tenant HPC clusters where users do not have root privileges. Singularity runs entirely in user space.
- **Resource management**: Singularity integrates naturally with SLURM's resource accounting, whereas Docker's own cgroup management can conflict with the cluster scheduler.
- **Filesystem access**: Singularity provides transparent access to shared parallel filesystems through bind mounts, without the additional complexity of Docker's overlay filesystem and network namespace.

A critical property of Singularity images is that they are *read-only* at runtime: the container's internal filesystem cannot be modified during execution. This immutability improves reproducibility by ensuring that each evaluation runs against an identical base environment, but it also introduces an important engineering constraint. The patched source code resides on the host filesystem and cannot simply be installed into the container through the usual *pip install* workflow.

Instead, the system bind-mounts the patched repository into the container and configures the Python import path so that the patched code takes precedence over the pre-installed packages available in the container's testbed directory.

The SWE-bench Docker images are converted to *.sif* files. The conversion is attempted through the local Docker daemon when available, and falls back to a direct registry pull on nodes where Docker is unavailable. Successfully converted images are cached on the shared filesystem, allowing later evaluations to reuse them without rebuilding the environment. Full implementation details of the conversion pipeline are provided in Appendix B.1.

5.4.3 Test Execution

After applying the candidate patch and preparing the containerized environment, the repository's test suite is executed inside the Singularity container. This process involves two stages: preparing the patched code for execution within the read-only container, and invoking the appropriate test framework.

Package Preparation

Because Singularity images are read-only, the patched source code cannot be installed directly into the container's package directories. Instead, the system bridges the gap between the host filesystem and the container through bind mounts and a carefully configured *PYTHONPATH*:

1. The patched repository is bind-mounted into the container at */workspace*.

2. Pre-compiled C extension files (*.so* shared objects) are copied from the container's */testbed* directory into */workspace*, preserving the original directory structure. This avoids the need to recompile Cython or C extensions from source, which would require exact compiler and library versions that may differ between environments. Some repositories also require additional handling for generated files not covered by this copy process.

The ordering of *PYTHONPATH* entries is critical. Workspace paths (*/workspace/src*, */workspace*, */workspace/lib*) are placed *before* the *.pip_packages* directory so that the patched source code always takes precedence over any bundled versions of the same packages that might be installed as transitive dependencies of testing tools.

Framework Detection and Test Invocation

SWE-bench includes repositories that use different testing frameworks. The system automatically detects the appropriate framework by inspecting the test identifier format: identifiers containing *::* separators indicate pytest, while identifiers following the pattern *test_name (module.Class)* indicate Django's built-in test runner.

The evaluation executes both *FAIL_TO_PASS* and *PASS_TO_PASS* tests provided by SWE-bench. The first category verifies that the patch resolves the reported issue, while the second acts as a regression check to ensure that previously working functionality is not broken.

A patch is considered to pass dynamic verification only if all tests in both categories produce their expected outcomes. Test execution occurs entirely within the containerized environment using the Python interpreter from the *testbed* conda environment, ensuring consistent dependency versions across all evaluations. The exact container invocation commands are provided in Appendix B.2.

5.4.4 Parallel Evaluation with SLURM

Evaluating large numbers of SWE-bench instances sequentially would be computationally expensive. To accelerate the process, the dynamic verification pipeline is designed to run as SLURM

array jobs on the UMD's HPC cluster.

Each SLURM array task evaluates a single SWE-bench instance. Tasks execute independently and use isolated working directories to prevent shared state conflicts.Table 5.4.2 summarizes the resource allocation per task.

Table 5.4.2: SLURM resource allocation per evaluation task.

Resource	**Allocation**
CPUs per task	4
Memory	8 GB
Wall-clock time limit	2 hours
GPU	Not required

Container images are cached on the shared filesystem, allowing multiple instances from the same repository to reuse the same container image. This significantly reduces the cost of environment preparation during large-scale evaluations.

5.4.5 From Dynamic to Semantic Verification

Dynamic verification ensures that a candidate patch satisfies the repository's existing test suite. However, passing these tests does not guarantee that the patch implements the intended behavioral change. A model-generated patch may pass all tests while implementing a different fix or introducing subtle behavioral changes that are not captured by the existing test suite.

To address this limitation, the next layer of the verification harness performs semantic verification. This stage extracts behavioral claims from the issue description, generates new tests targeting those claims, and executes them within the *same* Singularity containers built during the dynamic layer.

Reusing the containerized environment provides two important benefits. First, generated tests run against the exact same dependency versions and compiled extensions as the original test suite, eliminating discrepancies between verification stages. Second, the semantic layer can perform *discriminative testing*: each generated test is executed both against the patched code and against the original unpatched code, using the same containerized environment. A test that passes on the patched version but fails on the original version provides evidence that the patch implements the claimed behavioral change rather than passing trivially.

5.4.6 Limitations

Several limitations of the dynamic verification layer should be noted:

- **Test suite coverage.** The verification is bounded by the quality and coverage of the repository's existing tests. If the original test suite does not exercise the code paths affected by a patch, the dynamic layer cannot detect regressions in those areas.

- **Environment reproduction fidelity.** While SWE-bench Docker images provide near-exact environment reproduction, the Docker-to-Singularity conversion and the use of bind mounts (rather than running inside */testbed* directly) introduce minor differences. Certain projects with hard-coded paths or environment-sensitive configurations may require project-specific workarounds, as implemented for Astropy, Matplotlib, and pytest in our system.

- **Build failures.** A small fraction of instances fail during the containerization phase due to removed Docker images, authentication rate limits, or incompatible C extensions. These instances are excluded from evaluation rather than counted as failures.

- **Test flakiness.** Some repository tests exhibit non-deterministic behavior (e.g., timing-dependent assertions, network calls). The current system does not implement flaky-test detection or retry logic at the test level, which may introduce noise into the pass/fail signal.

5.5 Semantic Verification Layer

The semantic verification layer constitutes the third stage of the proposed multi-layer verification harness. While the static layer evaluates structural code quality and the dynamic layer validates candidate patches against the original test suite, the semantic layer aims to determine whether the patch actually satisfies the behavioral intent described in the issue report.

Test-suite-based verification alone is insufficient for validating automatically generated patches: a patch may satisfy existing tests without correctly addressing the underlying issue described in the bug report. This limitation has been observed both in the automated program repair literature and in recent analyses of SWE-bench-style evaluation protocols [3, 8].

Unlike the static verification layer, which assesses surface-level code properties (syntax, style, type consistency), the semantic layer reasons about the *behavioral intent* expressed in the issue description and produces executable test cases that discriminate between buggy and patched code. The layer proceeds in two main stages: (1) **claim extraction**, which derives structured, testable behavioral assertions from the issue description and grounds them in the patch diff; and (2) an **agentic test-generation loop**, which iteratively plans, generates, validates, and refines pytest test cases for each claim until a discriminative test is produced or a retry budget is exhausted. The pipeline is illustrated in Figure 5.5.3.

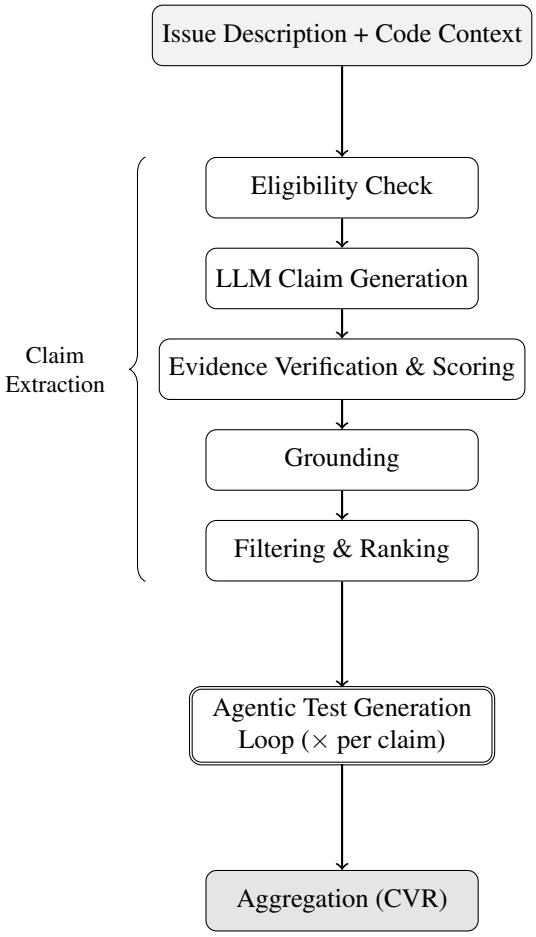

Figure 5.5.3: Semantic verification layer pipeline overview.

The objective of the semantic layer is therefore to transform the natural-language description of a bug report into executable behavioral checks that can discriminate between the buggy and the corrected program versions.

5.5.1 Claim Definition

A **claim** is a structured, testable behavioral assertion describing what the code *should do* after the bug is fixed. Each claim captures a single observable property of the patched program in a format amenable to automatic test generation. The claim schema consists of the following fields:

- **claim_id**: A unique identifier (e.g., `C1`, `C2`).
- **claim_type**: One of four behavioral categories:
  - `return` — the function produces a specific return value or type,

  - `exception` — the function raises or ceases to raise a specific exception,
  - `invariant` — a property that holds across inputs (e.g., idempotency, ordering),
  - `state_change` — an observable mutation of object or module state.

- **claim_text**: A one-sentence natural-language description of the expected behavior.
- **given / when / then**: A structured precondition–action–postcondition triple that specifies the test scenario concretely.
- **target_symbols**: A list of function, class, or attribute names that the claim references, drawn from the code context.
- **confidence**: A self-assessed extraction confidence level (`high`, `medium`, or `low`).
- **evidence**: Exact verbatim spans quoted from the issue text that support the claim.

Each claim represents a partial behavioral assertion derived from the issue description rather than a complete formal specification of the fix.

An example claim for a hypothetical issue reporting that `parse_date` silently returns `None` on invalid input instead of raising `ValueError` would be:

```
{
  "claim_id": "C1",
  "claim_type": "exception",
  "claim_text": "parse_date raises ValueError on invalid date strings",
  "given": "An invalid date string such as '2024-13-01'",
  "when": "parse_date('2024-13-01') is called",
  "then": "A ValueError is raised",
  "target_symbols": ["parse_date"],
  "confidence": "high",
  "evidence": {
    "spans": ["parse_date silently returns None for invalid dates"]
  }
}
```

Listing 5.2: Example claim in JSON format

The *given/when/then* triple is central to the design: it provides the test generator with concrete preconditions, an invocation target, and an observable postcondition, avoiding the ambiguity that typically degrades LLM-generated test quality. Table 5.5.3 shows an example of how a user prompt can be structured using the given/when/then decomposition derived from the original problem statement (see Listing C.1 in Appendix C).

The given/when/then structure was selected because it provides a simple behavioral specification

CLAIM COMPONENT	OUTPUT DESCRIPTION	RELEVANT ISSUE CONTEXT (CODE & OUTPUT)

GIVEN The initial condition or state before the action is taken.	Empty lists/arrays `[ ]`	`# Issue when passing empty lists/arrays...`
		`In [1]: from astropy.wcs import WCS`
		`In [2]: wcs = WCS('2MASS_h.fits')`

WHEN The action or event that triggers the behavior being tested.	Calling `wcs_pix2world` $f()$	`In [3]: wcs.wcs_pix2world([], [], 0)`
THEN The expected outcome or result of the action under the given condition.	Return empty array without triggering exceptions ✓	`InconsistentAxisTypesError Traceback ...` `~/Dropbox/Code/Astropy/astropy/wcs/wcs.py`

Table 5.5.3: Example decomposition of a bug report into a Given/When/Then claim structure.

format widely used in behavior-driven development, and it maps naturally to executable test scenarios.

5.5.2 Claim Extraction

The claim extraction pipeline transforms a raw issue description and its associated code context into a set of scored and filtered claims. The pipeline proceeds through the following stages.

Eligibility Check

Before invoking the LLM, the system performs a lightweight keyword-based check on the problem statement to determine whether it contains sufficient actionable information (e.g., error descriptions, reproduction steps, expected-behavior statements, or code snippets). Instances that fall below an actionability threshold are marked ineligible and skipped, avoiding unnecessary LLM calls on vague or underspecified issues.

LLM-Based Claim Generation

The core extraction step provides the LLM with the problem statement and the relevant source code context, and instructs it to produce a structured JSON array of claims following the schema defined in Section 5.5.1. The prompt emphasizes:

- concrete *given/when/then* triples with real function names,
- observable outcomes (return values, exceptions, state changes),

- exact evidence spans quoted verbatim from the issue,
- avoidance of vague assertions, hallucinated symbols, and over-specification.

The raw LLM output is parsed using a robust JSON recovery pipeline capable of handling common formatting errors in LLM-generated structured output. Each parsed claim is then validated against the required schema and normalized (e.g., coercing string-valued symbol lists to arrays, defaulting unrecognized confidence levels to `medium`).

Evidence Verification and Scoring

Each claim undergoes evidence verification against multiple sources. The system checks whether the claimed evidence actually spans in the issue text (exact or partial token overlap), whether they appear in code docstrings or comments, and whether the claim's target symbols appear in any stacktrace extracted from the issue. These signals are aggregated into a composite score that reflects both the traceability of the claim to the issue and the specificity of its behavioral assertion. Claims scoring below a minimum threshold are excluded from further processing.

Claim Filtering and Ranking

The final claim set consists of grounded claims (see Section 5.5.3) whose composite score meets the minimum threshold, sorted in descending score order. A configurable cap limits the number of claims forwarded to the test generation stage, balancing coverage against computational cost.

Filtering low-confidence or weakly grounded claims is important because poorly specified behavioral assertions tend to produce non-discriminative or unstable test cases.

5.5.3 Grounding Mechanism

Raw LLM-generated claims may reference symbols that do not exist in the codebase or that are unrelated to the patch. The **grounding** stage verifies that each claim's *target_symbols* can be traced back to concrete code artifacts, establishing a verifiable link between the natural-language assertion and the program under test. This step serves as a hallucination filter: claims whose target symbols cannot be located in the patch or its surrounding context are either discarded or downgraded.

Grounding is evaluated at three progressively weaker levels of confidence, summarized in Table 5.5.4.

Strong Grounding

The strongest form of grounding requires that a claim's target symbol corresponds to a program entity, function, class, constant, or instance attribute that is directly defined or modified in the patch diff. The system extracts candidate symbols from added and removed lines of the unified diff using pattern-based parsing, and stores each match together with auditable evidence (file path, line number, code snippet) for downstream traceability.

Level	Status	Criterion
3	`strong`	At least one target symbol matches a function, class, constant, or attribute that is *defined or assigned* in an added or removed line of the patch diff.
2	`weak_file`	The symbol appears somewhere in the full text of a file touched by the patch, but not directly in a diff line.
1	`weak_ref`	The symbol appears in the broader code context (e.g., import statements, surrounding function calls).
0	`none`	No match is found at any level.

Table 5.5.4: Grounding strength levels, ordered from strongest to weakest.

Symbol Normalization

Because issue descriptions, code, and LLM output may use inconsistent naming conventions, all symbol comparisons are performed over a normalized variant set. Given a symbol string, the normalizer generates variants by splitting dotted names (e.g., `Class.method` produces both `Class` and `method`), stripping `self`/`cls` prefixes, and converting between `camelCase` and `snake_case`. A match is declared when the variant sets of the target and diff symbols intersect, while common built-in names are excluded to prevent spurious matches.

Weak Grounding Fallbacks

If no strong match is found, the system falls back to progressively weaker checks: first searching the full text of files touched by the patch (`weak_file`), then the broader code context supplied to the extraction prompt (`weak_ref`). Claims that receive a grounding status of `none` are discarded by default.

5.5.4 Test Generation (Agentic Loop)

Because the generation of reliable tests from natural-language claims is a challenging task, a single LLM generation step is often insufficient. The system therefore adopts an agentic refinement loop that iteratively improves test generation using execution feedback.

For each extracted claim, an **agentic closed loop** (See Figure 5.5.4) iteratively generates, validates, and refines a pytest test case. The loop runs for up to K attempts (default $K = 3$) and terminates early on success or when it detects an unrecoverable failure mode.

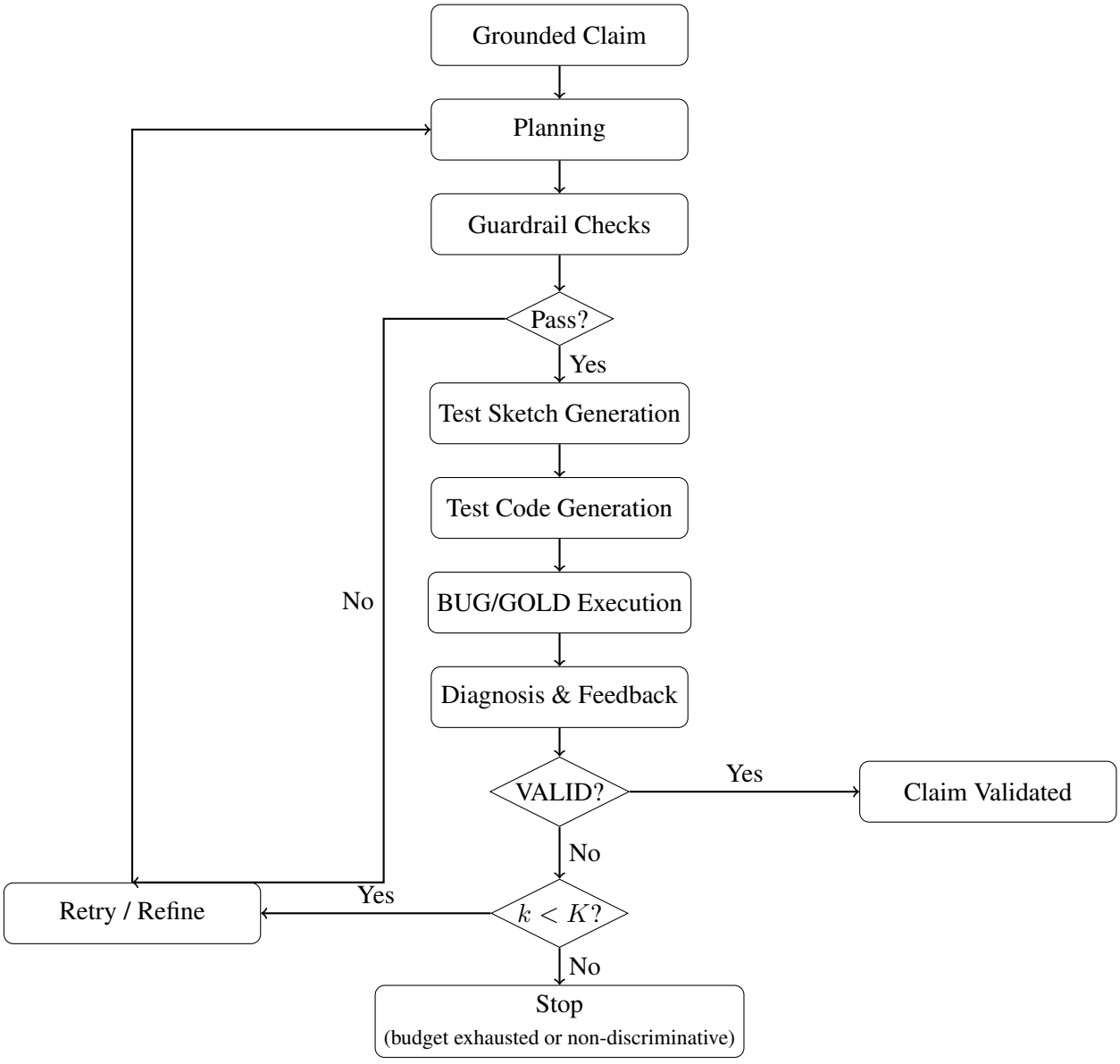

Figure 5.5.4: Agentic closed loop for semantic test generation

Loop Architecture

Each iteration of the loop proceeds through six phases:

1. **Planning**: A plan is assembled from the claim context (target module, symbols, signatures, grounding evidence, issue context) and refined using feedback from any previous failed iterations.

2. **Guardrail validation**: The plan is evaluated against pre-generation checks designed to

detect likely failure modes (e.g., missing imports, incorrect signatures) before invoking the LLM.

3. **Test sketch generation**: The LLM produces a structured test design specifying the strategy, assertions, fixtures, data setup, and edge cases, without writing code.

4. **Test code generation**: A second LLM call converts the sketch, plan, and guardrail context into a standalone pytest test file.

5. **Dual-environment verification**: The generated test is executed against both the buggy and patched versions of the repository in isolated containers.

6. **Diagnosis and feedback**: The verification outcome is classified, and structured feedback is extracted for the next iteration.

The planning stage operates in two layers. A **static planner** constructs the initial plan from the claim context, including a strategy for exercising the target symbols, the expected inputs derived from the *given/when/then* triple, observable postconditions to assert on, environment preconditions, and fallback strategies.

A **dynamic planner** refines the plan after each failed iteration. It receives the previous iteration's failure diagnosis and selects a *playbook*, a predefined set of corrective instructions keyed by the diagnosis category. For example, a signature-related failure triggers a playbook that instructs the LLM to use package-level imports instead of deep submodule paths. The refinement is additive: playbook entries are appended to the existing plan without duplicating information already present.

Guardrail Checks

Before generating test code, the plan is evaluated against a suite of guardrail checks that aim to detect problems early:

- **Import check**: Verifies that the target module can be resolved. This check is non-blocking, since many project-specific modules require dependencies only available in the execution container.

- **Signature check**: Verifies that the target symbols exist in the module and extracts their call signatures, including required arguments and instance-method detection.

- **Fixture check**: Identifies available pytest fixtures from the repository's test infrastructure, producing advisory notes for the test generator.

- **Probe check**: For zero-argument callables, performs a safe trial invocation to verify basic callability.

When a blocking guardrail fails, the loop skips test generation for that iteration and routes the

failure through the dynamic planner. A stuck-loop detector terminates early if the same failure repeats across consecutive iterations.

Two-Stage Test Generation

Test generation is split into two LLM calls to separate design from implementation:

1. **Test Sketch**: The first call produces a structured plan (strategy, assertion targets, required fixtures, concrete data setup, edge cases, and a compliance checklist). This intermediate representation allows guardrail context and previous feedback to shape the test design before any code is written.

2. **Test Code**: The second call receives the sketch, the full plan, guardrail context, and on retry iterations, the previous failing code and detected anti-patterns. It produces a self-contained pytest test file that follows the *given/when/then* structure of the claim.

Dual-Environment Verification

The generated test is executed in two isolated Singularity containers:

- **BUG environment**: The repository at the base commit, *without* the patch applied.

- **GOLD environment**: The repository with the patch applied.

Both environments undergo the same setup process: the claim test is copied into the repository, project dependencies are installed, and pytest is invoked. The comparison of outcomes across these two environments is what determines whether the test is discriminative.

Verification Classification

The BUG and GOLD outcomes are compared to assign a classification label, as summarized in Table 5.5.5.

A claim-test is considered **successful** only when it receives the `VALID` label, meaning the test fails on the buggy code and passes on the patched code. This criterion ensures that the test captures a behavioral difference that is directly attributable to the patch.

This outcome indicates that the generated test reproduces the original faulty behavior while confirming that the patch eliminates it.

Failure Diagnosis and Feedback Loop

When a test does not achieve the `VALID` label, a diagnostics module classifies the failure into an actionable category (e.g., import errors, signature mismatches, overconstrained assertions, environment issues). Additionally, a pattern detector analyzes the generated test code for common anti-patterns such as imports from private modules or incorrect fixture usage.

BUG	GOLD	Label	Interpretation
FAIL	PASS	`VALID`	Test discriminates: it reproduces the bug and confirms the fix.
PASS	PASS	`NON_DISCRIMINATIVE`	Test passes on both versions; it does not detect the bug.
FAIL	FAIL	`OVERCONSTRAINED`	Test is too strict; it fails even on the fixed code.
PASS	FAIL	`INVERTED`	Test logic is inverted: the fix causes a regression.
ERROR/TIMEOUT	*	`UNRESOLVED`	Execution error or timeout; inconclusive.

Table 5.5.5: Claim-test verification labels based on BUG vs. GOLD execution outcomes.

The diagnosis category, error details, and detected anti-patterns are composed into a structured feedback signal that is passed to the dynamic planner and subsequently to both LLM calls in the next iteration. This creates a *closed feedback loop*: each failed attempt narrows the search space by ruling out specific failure modes and providing the LLM with concrete error output from the actual execution environment.

The loop terminates under three conditions:

1. **Success**: The test receives the `VALID` label.

2. **Non-discriminative exit**: The test is classified as `NON_DISCRIMINATIVE`, indicating that the claim itself may not capture a behavioral difference introduced by the patch.

3. **Budget exhaustion**: The maximum number of attempts K is reached, or the loop detects that it is stuck in the same failure mode across consecutive iterations.

5.5.5 Semantic Layer Aggregation

After the agentic loop has processed all claims for a given instance, the semantic layer aggregates the results into a summary. The primary metric is the **Claim Validation Rate (CVR)**, defined as the fraction of extracted claims for which we can generate a discriminative test that fails on BUG and passes on GOLD.

$$\text{CVR} = \frac{|\{\text{claims with label } \texttt{VALID}\}|}{|\{\text{total claims processed}\}|} \tag{5.1}$$

A higher CVR indicates that a larger proportion of extracted claims can be validated through discriminative tests, suggesting that the patch successfully resolves the behavioral issues described

in the bug report.

The summary also reports the full label distribution across claims (how many were classified as `VALID`, `NON_DISCRIMINATIVE`, `OVERCONSTRAINED`, etc.) and the extraction-stage statistics (grounding rate, average score, specificity rate). Together with the Static Quality Index from the static layer, the CVR provides a complementary behavioral signal: the static layer assesses code quality in isolation, while the semantic layer assesses whether the patch actually addresses the reported issue.

5.6 User Application

The verification harness described in the preceding sections operates primarily as a batch-oriented pipeline, processing SWE-bench instances non-interactively and producing structured JSON outputs for offline analysis. While this execution mode is suitable for large-scale evaluation, it provides limited support for inspecting intermediate artifacts, diagnosing individual failures, or comparing alternative patches for the same issue.

To address these limitations, an interactive user application was developed that exposes the full three-layer verification pipeline through a graphical interface. The application enables users to select a problem instance, choose a patch (either the gold reference solution or an LLM-generated candidate), execute the verification pipeline on demand, and inspect detailed results at each stage.

This component serves a dual purpose. First, it provides a practical tool for qualitative analysis, complementing the quantitative results presented in Chapter 6.1 and subsequent sections. Second, it demonstrates that the proposed verification harness is modular and composable, supporting both batch evaluation and interactive exploration without modification to its core logic.

5.6.1 Design Goals

The application is designed around three core principles:

1. **Transparency**: All intermediate artifacts produced by the pipeline, including extracted claims, grounding evidence, generated test cases, execution logs, and classification labels, are exposed to the user. This ensures that every verification outcome can be traced back to its underlying evidence.

2. **Unified execution**: The same pipeline implementation is used for both gold patches and candidate patches. The only varying input is the patch diff, ensuring that comparisons between reference and generated solutions are methodologically consistent.

3. **Lightweight deployment**: The application runs on a single compute node alongside the vLLM inference server used by the semantic layer, requiring no additional external services beyond container images and model weights.

5.6.2 Architecture

The application is implemented as a Streamlit [40] web interface that interacts with the verification pipeline and a co-located vLLM [31] server. Figure 5.6.5 illustrates the deployment architecture.

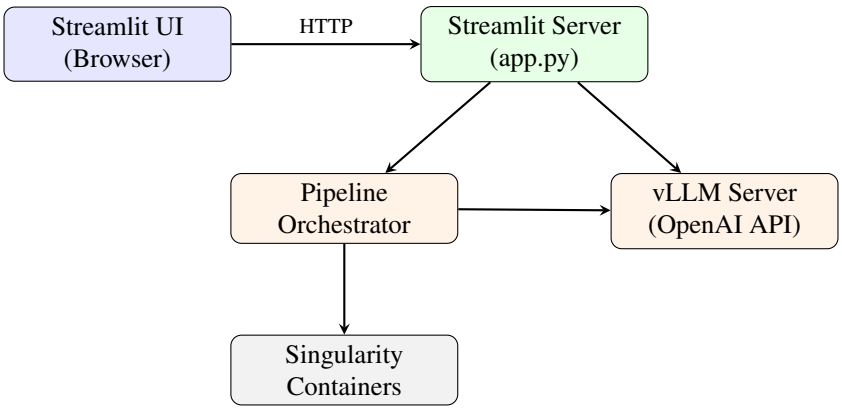

Figure 5.6.5: Deployment architecture of the interactive application.

A single SLURM job provisions the required resources (CPU, GPU, and memory) and launches both the vLLM server and the Streamlit interface. The vLLM server exposes an OpenAI-compatible API used by the semantic layer for claim extraction and test generation, while the pipeline orchestrator coordinates execution across all verification layers. Containerized test execution is handled through pre-built Singularity images, ensuring reproducibility and consistency with the batch pipeline.

5.6.3 Interactive Workflow

The primary interaction mode of the application focuses on SWE-bench instances and follows a structured workflow:

Instance Selection. Only instances with available container images and prior successful semantic validation are presented to the user. This filtering ensures that all selectable instances can complete the full pipeline without runtime failures.

Patch Selection Users may choose between the gold patch and multiple candidate patches generated during prior experiments. Candidate patches are annotated with their SWE-bench status (*resolved* or *failed*), enabling immediate contextual interpretation.

Pipeline Execution Users can selectively enable the static, dynamic, and semantic layers (as seen in Figure 5.6.6) and trigger execution. Each layer reports progress incrementally, providing real-time feedback:

- **Static layer**: evaluates syntax, structure, and code quality metrics.
- **Dynamic layer**: executes the repository test suite within a containerized environment.

- **Semantic layer**: extracts behavioral claims, performs grounding, and iteratively generates discriminative test cases.

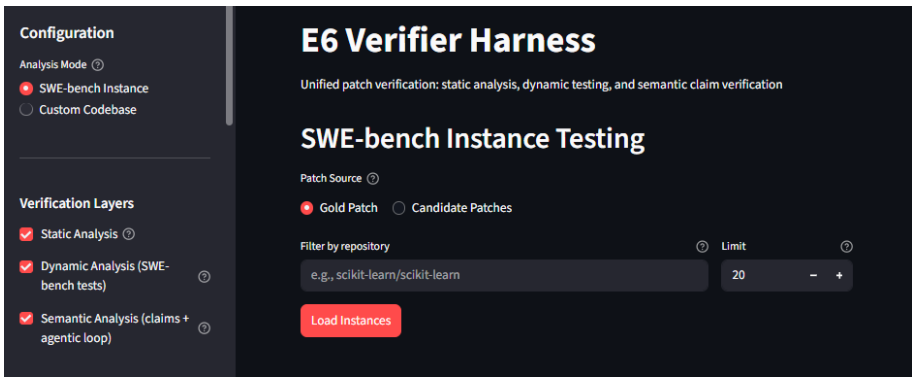

Figure 5.6.6: Streamlit app (Instance and Verification Layer selection)

Results Visualization Results are presented through a hierarchical interface. A top-level dashboard summarizes key metrics (e.g., Static Quality Index, test pass rate, claim validation rate), while detailed views expose per-layer outputs. Figure 5.6.7 shows how results are presented in the app, with each layer analyzed, in this case, the static analysis output.

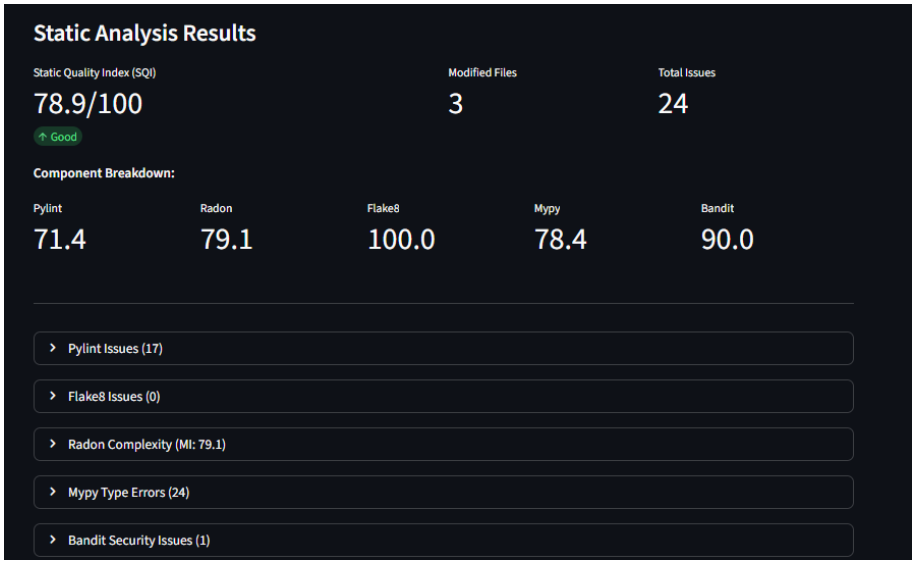

Figure 5.6.7: Static Layer Results displayed on the Streamlit app

In the semantic layer, claims are grouped by outcome (`VALID` vs. failed classifications), with expandable panels providing access to generated tests, execution traces, and classification rationales. This design enables both rapid assessment and fine-grained inspection.

Figure 5.6.8: Candidate patch selection and display on the Streamlit app

5.6.4 Candidate Patch Analysis

A central capability of the application is the analysis of LLM-generated candidate patches (Figure 5.6.8). Unlike the standard SWE-bench evaluation, which relies solely on test-suite outcomes, the application provides a richer diagnostic view by combining static, dynamic, and semantic signals.

In particular, candidate patches that pass the original test suite may still fail semantic verification, revealing incomplete or incorrect implementations of the intended behavior. Conversely, patches that fail the test suite may satisfy some extracted claims, indicating partial correctness.

By enabling side-by-side comparison between gold and candidate patches under identical verification conditions, the application supports detailed analysis of LLM failure modes, including incomplete fixes, overfitting to test cases, and unintended regressions in code quality.

5.7 Project Planning and Progression

This project follows a progressive and modular development strategy in which each layer is implemented independently and then integrated into a unified pipeline. Although the tasks are presented sequentially in the timeline, the development process followed an iterative approach inspired by *Agile* principles. This allowed continuous refinement of the system, enabling updates and adjustments to different components as intermediate results were obtained.

Figure 5.7.9 presents the full progression of the project over six months, structured into four main development blocks. The timeline reflects both the sequential construction of the verification harness and the iterative refinement of its components.

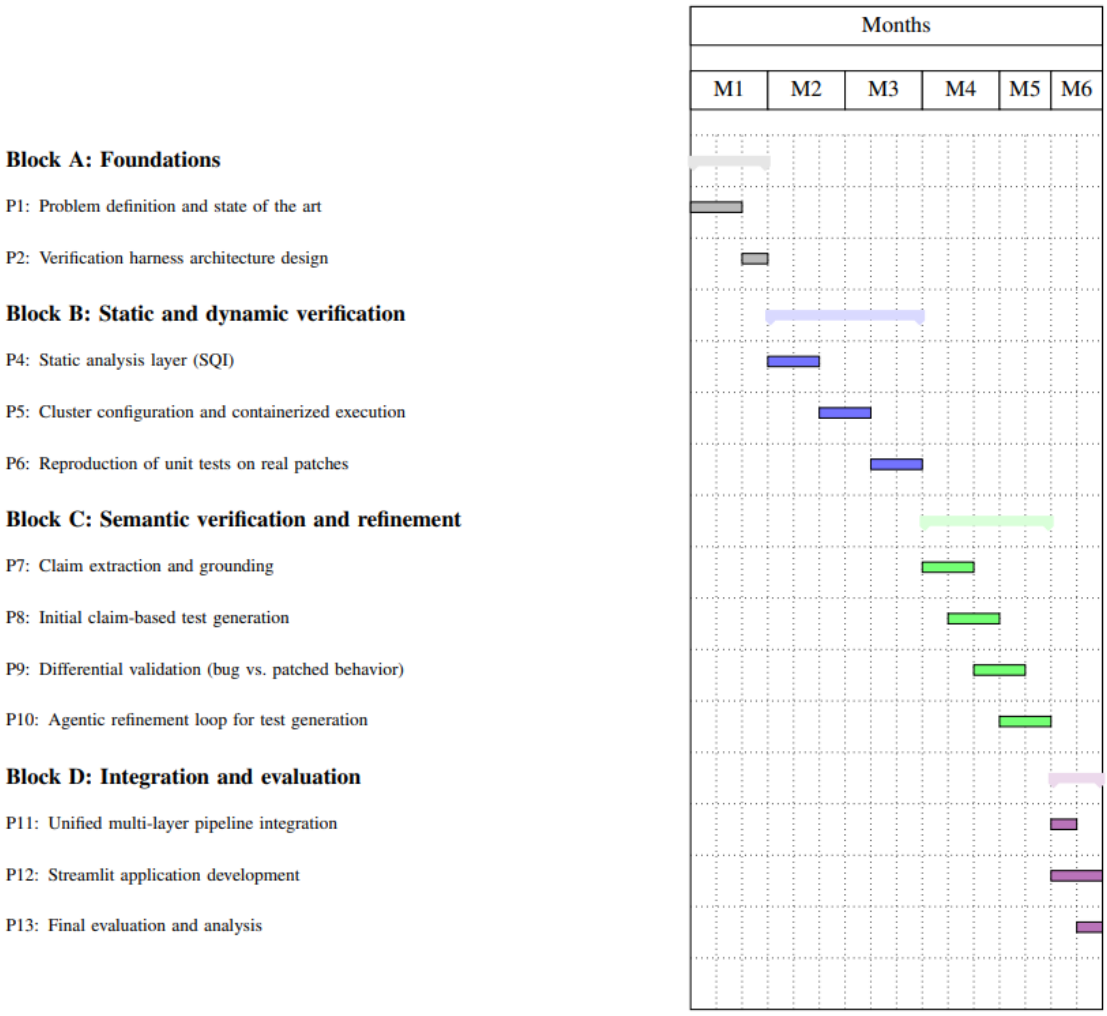

Figure 5.7.9: Project timeline over the six-month development period.

Block A: Foundations and System Design. The initial phase focuses on problem definition, literature review, and the design of the multi-layer verification architecture. During this stage, the limitations of test-suite-based evaluation were analyzed, motivating the need for a multi-layer approach combining static, dynamic, and semantic verification.

Block B: Static and Dynamic Verification. The second phase involves the implementation of the first two verification layers. This includes the development of the static analysis pipeline (syntax validation, structural metrics, and Static Quality Index) and the setup of the dynamic verification layer through containerized execution of repository test suites. At the end of this stage, a fully functional baseline pipeline is established.

Block C: Semantic Verification and Iterative Refinement. The third phase focuses on the semantic layer. Initial experiments with direct claim-based test generation revealed limitations in producing sufficiently discriminative tests. These observations motivated the development of

an agentic refinement loop, in which test generation is guided by execution feedback and iterative improvement strategies. This phase represents the main research contribution of the project.

Block D: Evaluation and Analysis. The final phase is dedicated to evaluation and consolidation. This includes large-scale experimentation on SWE-bench Lite, comparison of different model configurations, analysis of verification outcomes across layers, and the study of failure modes in LLM-generated patches. The results obtained in this phase form the basis of the evaluation presented in the next chapter.

Chapter 6

Result Analysis

This chapter presents the experimental results obtained by applying the multi-layer verification harness described in Chapter 5. The static layer is evaluated on the full SWE-bench Lite dataset ($N = 300$), while the dynamic and semantic layers are evaluated on a subset of 18 instances. This subset is constrained by the availability of pre-built execution environments, which are limited due to storage and reproducibility constraints in the shared computing infrastructure. As a result, end-to-end results apply only to this subset and should be interpreted accordingly.

As previously mentioned, experiments were executed on the University of Maryland *Nexus* computing cluster using SLURM array jobs with the resource configuration described in Section 5.4 (Table B.3). The evaluation dataset consists of $N = 300$ SWE-bench Lite instances spanning 12 open-source Python repositories.

Figure 6.0.1 summarizes experiment coverage by layer.

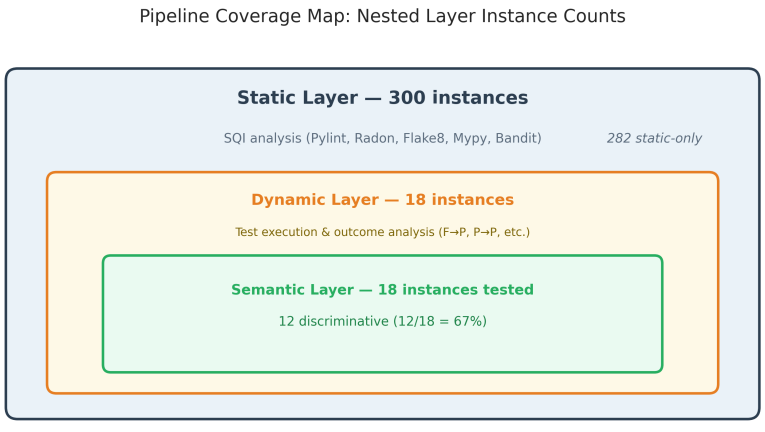

Figure 6.0.1: Pipeline coverage across verification layers.

This coverage disparity has important implications for the interpretation of the results. While the static layer provides a comprehensive view of the dataset, the dynamic and semantic layers operate on a constrained subset of instances. As a result, findings related to execution outcomes and discriminative test generation should be interpreted as case-study evidence rather than statistically representative estimates of the full benchmark.

Furthermore, the semantic layer inherits the constraints of the dynamic layer, as it requires executable environments to validate generated tests.

6.1 Static Verification Layer

This section evaluates the behavior of the proposed static verification layer when applied to the 300 gold patches of SWE-bench Lite. The goal of this experiment is to assess whether the static layer provides a meaningful first-pass quality signal for candidate patches while remaining computationally efficient.

For each instance, the system reconstructs the repository at its `base_commit`, applies the gold patch, and executes the complete static analysis pipeline, including syntax validation, structural metric extraction, and computation of the Static Quality Index (SQI). Because SWE-bench patches typically modify only a small portion of large source files, all analyzers operate under a patch-scoped filtering strategy that attributes findings only to the lines modified by the patch.

6.1.1 Experimental Setup

Each SWE-bench instance was processed as an independent SLURM array task (`--array=0-299`) using 4 CPU cores and 8 GB of memory. The pipeline performs the following steps:

1. Clone the repository and check out the `base_commit`.
2. Apply the gold patch using `git apply`.
3. Execute the five static analyzers (Pylint, Flake8, Radon, Mypy, Bandit) on each modified Python file.
4. Filter analyzer findings to retain only issues located on patch lines (patch-scoped filtering).
5. Compute the Static Quality Index (SQI) from normalized component scores.
6. Perform structural analysis (AST validation, function extraction, and AST diff computation).
7. Serialize results to a per-instance JSON report.

All 300 instances completed successfully. The average execution time per instance was 13.1 sec-

onds (median 12.1 seconds, maximum 55.4 seconds), confirming that the static layer provides a lightweight verification step suitable for early-stage filtering.

6.1.2 Patch-Scoped Analysis

A central design choice of the static verification layer is the scope of measurement. SWE-bench patches are typically small and localized modifications. Across the dataset, the median patch modifies only 13 lines of code, while the median file size is 803 lines.

If static analyzers were applied at the file level without filtering, pre-existing technical debt in the surrounding codebase would dominate the results and obscure the quality of the patch itself. To address this issue, all analyzer outputs are filtered to retain only findings occurring within the diff hunks of the patch.

Table 6.1.1 shows the impact of patch-scoped filtering across the dataset.

Analyzer	File-Scope Issues	Patch-Scope Issues	Attributable (%)
Pylint	33 417	543	1.6%
Flake8	10 935	253	2.3%
Mypy	70 008	1 827	2.6%

Table 6.1.1: Issue counts before and after patch-scoped filtering.

Only 1.6–2.6% of reported issues occur on patch lines, demonstrating that most static analysis warnings originate from legacy code rather than from the patch itself. This result highlights the importance of patch-scoped evaluation when analyzing candidate patches.

6.1.3 Syntax Validity

All 300 gold patches produced syntactically valid Python files according to AST parsing. While this result is expected for human-written patches merged into production repositories, it validates the usefulness of the syntax gate as a fail-fast filter: syntactically invalid patches can be rejected before engaging more expensive verification stages.

6.1.4 Structural Characteristics

Table 6.1.2 summarizes the structural properties of the modified files.

These results confirm that SWE-bench patches are typically narrow and surgical modifications. The median patch modifies a single function within a file containing 41 functions, resulting in a very low AST diff ratio.

6.1.5 Static Quality Index

Figure 6.1.2 shows the distribution of patch-scoped SQI values across all 300 instances.

Metric	Mean	Median	Min	Max
Functions per file	64.3	41	0	307
Classes per file	6.3	3	0	40
AST nesting depth	14.2	14	5	25
Avg. function length (LOC)	19.3	15.5	0.0	99.0
Changed functions per patch	1.7	1	0	12
AST diff ratio	0.073	0.033	0.0	1.0
Patch LOC	17.8	13	6	73
File LOC	1 216	803	25	8 245

Table 6.1.2: Structural characteristics of modified files across the dataset.

The mean SQI is 83.1, with a median of 85.5 and a standard deviation of 11.3. The distribution is concentrated in the $[80, 90)$ range, indicating that most gold patches exhibit high code quality on the lines they modify.

Table 6.1.3 shows the classification distribution.

Classification	SQI Range	Count	Percentage
Excellent	≥ 85	153	51.0%
Good	$[70, 85)$	110	36.7%
Fair	$[50, 70)$	31	10.3%
Poor	< 50	6	2.0%

Table 6.1.3: SQI classification distribution.

Overall, 87.7% of patches fall into the *Good* or *Excellent* categories. This confirms that human-written patches generally maintain high code quality on the lines they modify.

6.1.6 Repository-Level Analysis

Figure 6.1.3 compares SQI distributions across repositories.

Once patch-scoped filtering is applied, the differences between repositories become relatively small. Even repositories with historically large codebases such as Django and SymPy maintain mean SQI values within the Good range, suggesting that quality differences observed under file-scoped analysis are primarily driven by legacy code rather than by patch quality.

6.1.7 Patch Size Sensitivity

Figure 6.1.4 examines the relationship between patch size and SQI.

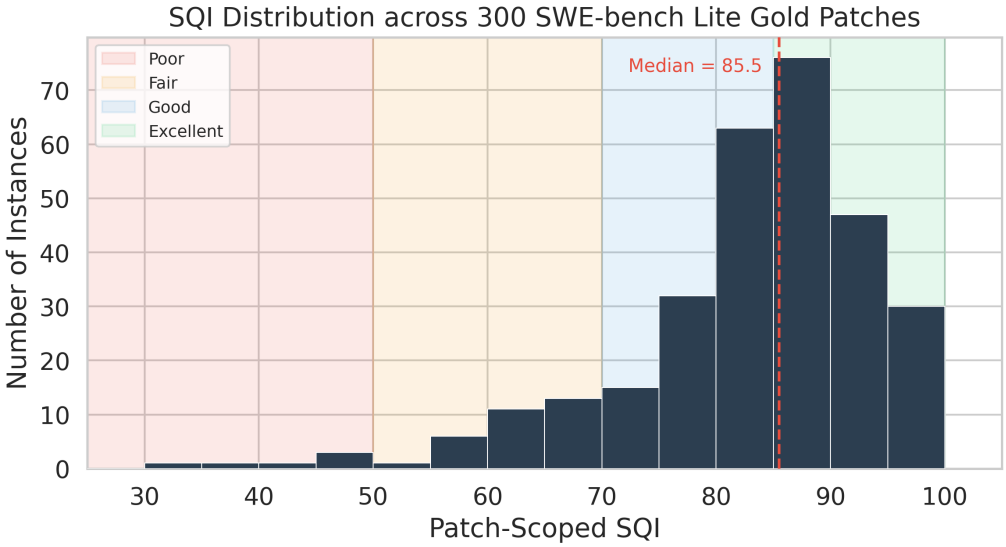

Figure 6.1.2: Distribution of the patch-scoped Static Quality Index across the 300 SWE-bench Lite gold patches.

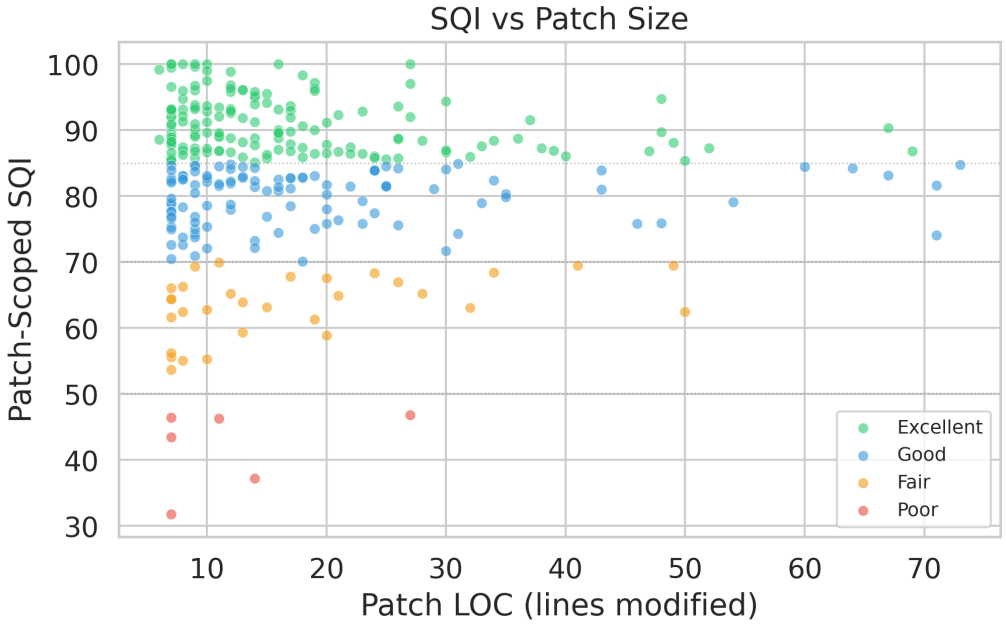

Figure 6.1.4: Relationship between patch size and SQI.

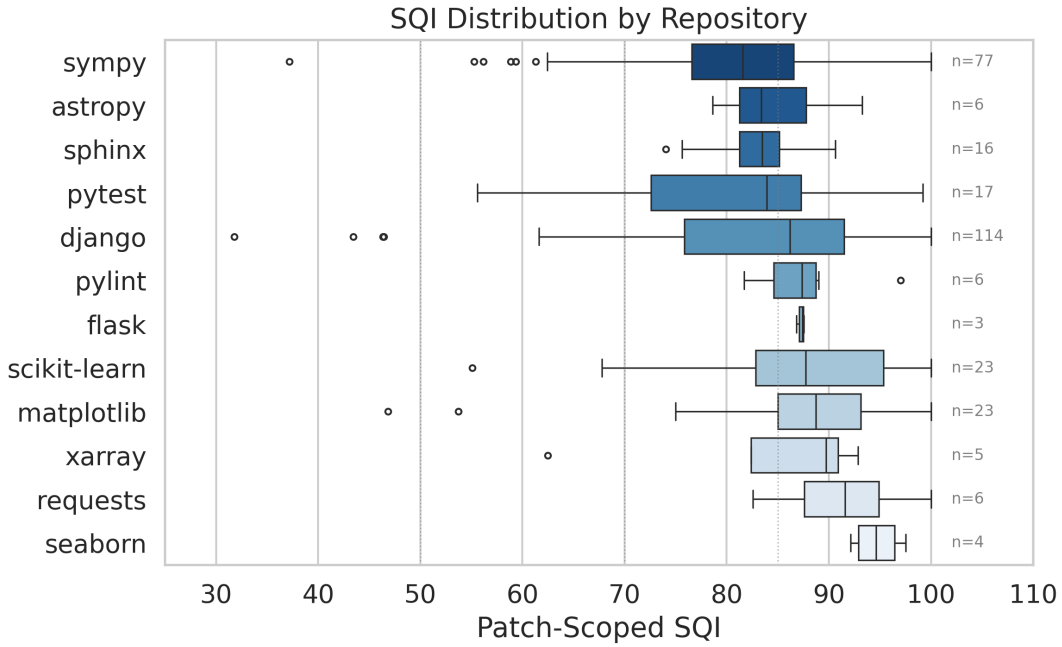

Figure 6.1.3: Distribution of SQI values grouped by repository.

Very small patches exhibit higher score variance because a single static analysis warning can have a disproportionate impact on the normalized score. This explains why most low-SQI instances correspond to patches modifying fewer than 15 lines.

6.1.8 Summary of Findings

The evaluation of the static verification layer yields several key findings:

- All 300 gold patches pass syntax validation.
- Patch-scoped filtering removes over 97% of analyzer findings that originate from legacy code rather than the patch.
- The mean SQI is 83.1, with 87.7% of patches classified as *Good* or *Excellent*.
- Pylint contributes most strongly to SQI variability.
- Very small patches are more sensitive to individual warnings.

While the static layer provides an efficient quality signal, it cannot verify whether a patch correctly resolves the underlying bug. For this reason, patches that pass static verification proceed to the dynamic verification layer, where they are evaluated using the repository's test suite.

6.2 Dynamic Verification Layer

This section evaluates the dynamic verification layer by executing the repository's own unit tests against the 18 gold patches for which pre-built Singularity images are available. The goal is to verify that the gold patch resolves the targeted failing behavior (FAIL_TO_PASS) without introducing regressions (PASS_TO_PASS).

6.2.1 Experimental Setup

Each instance was processed as an independent SLURM array task using 4 CPU cores and 8 GB of memory, matching the resource allocation of Table 5.4.2. The pipeline follows the containerized execution approach described in Section 5.4.2 and Appendix B:

The 18 evaluated instances span 7 repositories: Django (4), SymPy (4),Pylint (3), Pytest (3), Scikit-learn (2), Astropy (1), and Requests (1).

6.2.2 Dataset Characteristics

Upon execution, each instance produces a structured execution report containing detailed test statistics, including the number of test specifications, executed tests, pass/fail outcomes, and runtime information.

Across the 18 evaluated instances, the dynamic layer processes more than 400 FAIL_TO_PASS test specifications and several hundred PASS_TO_PASS regression tests. The number of test specifications per instance varies substantially, ranging from 1 to 355.

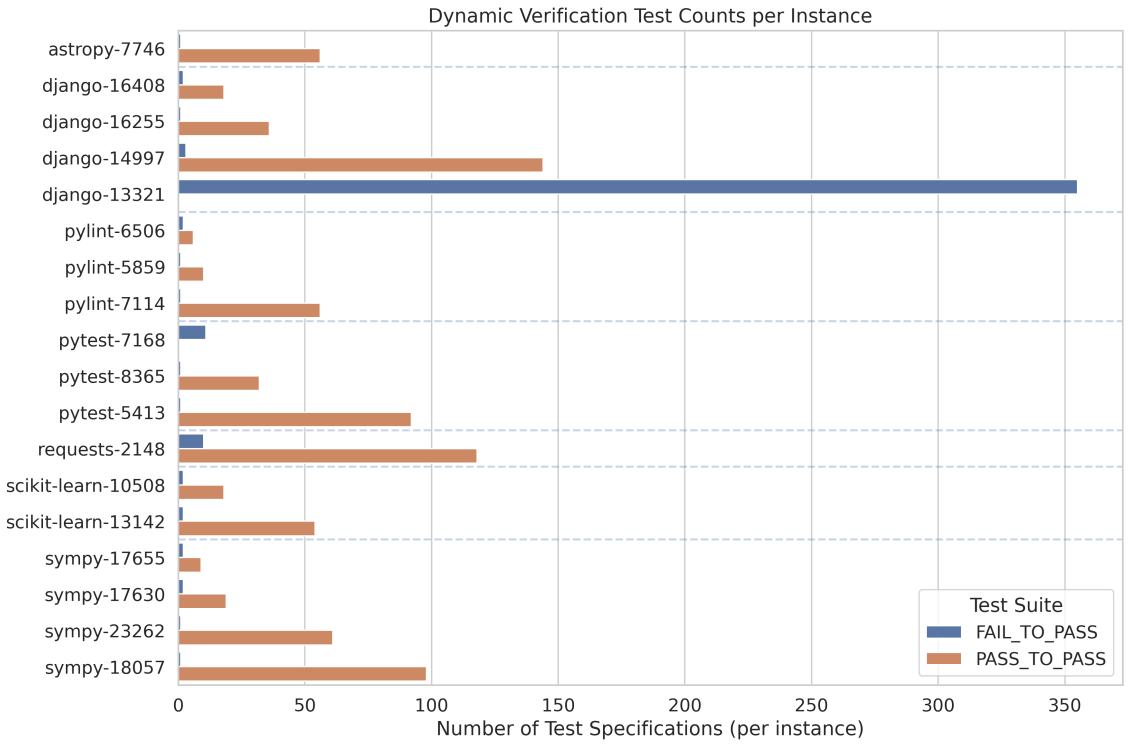

Figure 6.2.5: Dynamic verification test workload per instance.

Figure 6.2.5 shows the number of FAIL_TO_PASS and PASS_TO_PASS test specifications per instance. Most instances involve a relatively small number of failing tests, while a few cases, most notably `django-13321`, exhibit substantially larger test suites. This variability reflects the heterogeneous complexity of the benchmark and directly impacts execution cost, contributing to the runtime differences observed in the dynamic verification layer.

Due to test parametrization in frameworks such as Django and SymPy, the number of executed tests can exceed the number of test specifications. This distinction is captured in the execution reports through separate fields for test counts and executed test outcomes, allowing a more accurate characterization of effective test coverage.

6.2.3 Resolution Results

Table 6.2.4 summarizes the per-repository resolution rates.

All 18 instances pass their FAIL_TO_PASS tests, confirming that the targeted failing behaviors are resolved under the repository test suites. 17 of the 18 instances also pass all PASS_TO_PASS tests, yielding a full resolution rate of 94.4%.

The single unresolved instance (`pylint-dev/pylint-7114`) passes its FAIL_TO_PASS test

Repository	Instances	Resolved	Avg. Duration
Astropy	1	1/1	36 s
Django	4	4/4	107 s
Pylint	3	2/3	36 s
Pytest	3	3/3	9 s
Requests	1	1/1	28 s
Scikit-learn	2	2/2	29 s
SymPy	4	4/4	121 s
Total	**18**	**17/18 (94.4%)**	**67 s**

Table 6.2.4: Dynamic verification resolution rate per repository.

but exhibits 3 PASS_TO_PASS failures caused by environment-specific discrepancies under bind-mount execution (Section 5.4.6), rather than by the patch logic itself.

6.2.4 FAIL_TO_PASS Analysis

The FAIL_TO_PASS suite provides the primary positive verification signal of the dynamic layer: tests that fail in the buggy version must pass after the patch is applied. Across all 18 instances, all FAIL_TO_PASS tests pass successfully, confirming that the expected failing behavior is resolved in all cases.

The number of FAIL_TO_PASS test specifications per instance ranges from 1 to 355, with a median of 2. The instance with 355 specifications (`django-13321`) covers session handling across multiple backend implementations, all of which pass after the gold patch is applied.

Due to test parametrization, the number of executed tests can exceed the number of test specifications, as reflected in the difference between recorded test counts and executed test cases in some instances.

6.2.5 PASS_TO_PASS Analysis

The PASS_TO_PASS suite verifies that the patch does not introduce regressions. Of the 16 instances with non-empty PASS_TO_PASS suites, 16 pass all tests. The remaining instance (`pylint-7114`) fails 3 out of 56 PASS_TO_PASS tests due to environment-specific issues: two relative-import tests whose resolution differs under bind-mount execution and one test that expects a filesystem path absent from the container.

No runtime errors are observed in any instance, indicating that all failures correspond to logical discrepancies rather than execution crashes.

6.2.6 Execution Stability

The dynamic layer exhibits stable execution across all evaluated instances. No runtime errors are observed in either FAIL_TO_PASS or PASS_TO_PASS executions, indicating that all test suites run

successfully within the containerized environment.

This stability is particularly relevant given the heterogeneity of the evaluated repositories and the use of bind-mounted execution environments. The absence of execution failures confirms that the container preparation and dependency handling mechanisms are robust across diverse project configurations.

6.2.7 Execution Performance

The average execution time per instance is 67 seconds (median 36 s, maximum 253 s). Django and SymPy instances exhibit the longest execution times due to larger test suites and heavier framework initialization. Pytest instances complete in under 10 seconds, benefiting from lightweight test discovery.

Compared to the static layer (mean 13 s), the dynamic layer incurs approximately a $5\times$ increase in runtime, which is expected given the overhead of container startup, test framework initialization, and test execution. Nevertheless, all instances complete well within the 2-hour SLURM time limit.

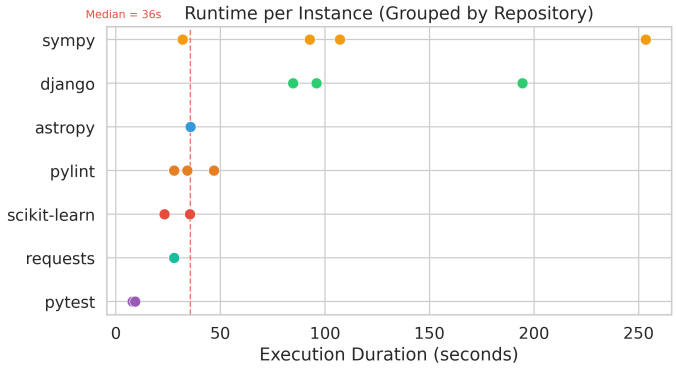

Figure 6.2.6: Dynamic verification runtime per instance grouped by repository.

Figure 6.2.6 shows the distribution of execution times across instances. Each point corresponds to a single benchmark instance, and the dashed line indicates the median runtime (36 seconds). The figure highlights substantial variability across repositories, with frameworks such as Django and SymPy exhibiting longer runtimes due to larger and more complex test suites, while lightweight frameworks such as Pytest complete significantly faster.

6.2.8 Summary of Findings

The evaluation of the dynamic verification layer yields the following key findings:

- All 18 gold patches pass their FAIL_TO_PASS tests, confirming correctness with respect to the provided test suites.

- 17 of 18 instances (94.4%) are fully resolved, with the single partial failure attributable to environment-specific execution limitations rather than the patch itself.
- The evaluation covers over 400 FAIL_TO_PASS test specifications and several hundred regression tests across diverse repositories.
- The dynamic layer provides a reliable execution-based filtering stage: patches that do not resolve the targeted tests, or that introduce regressions, can be rejected before proceeding to semantic verification.
- Mean execution time is 67 seconds per instance, making the dynamic layer feasible as an intermediate verification step between lightweight static analysis and heavyweight semantic evaluation.

Patches that pass both the static and dynamic layers proceed to the semantic verification layer, where LLM-generated tests evaluate whether the patch addresses the semantic claims described in the bug report.

6.3 Semantic Verification Layer

This section presents the experimental results of the semantic verification layer, whose design and implementation were described in Section 5.5. The evaluation consists of three stages: (1) claim extraction from issue descriptions, (2) agentic closed-loop test generation, and (3) validation of generated tests through dual execution on buggy and patched code.

6.3.1 Experimental Setup

The semantic verification layer was evaluated on the same 18 SWE-bench Lite instances used for dynamic verification, spanning 7 repositories. For each instance, the pipeline performs claim extraction, grounding, and iterative test generation using locally hosted LLMs served via vLLM.

Four quantized open-weight models were evaluated for both claim extraction and test generation:

- *Meta-Llama-3.1-70B-Instruct-AWQ-INT4*
- *Qwen2.5-72B-Instruct-AWQ*
- *Qwen2.5-Coder-32B-Instruct-AWQ*
- *Qwen3-32B-AWQ*

Combining these models yields 16 configurations. The agentic loop was executed with a maximum budget of $K = 10$ attempts per claim, resulting in a total of 412 experiment runs. A run represents

one execution of the agentic test-generation loop for one grounded claim of one instance, using a specific Test Generator/Claim Extractor model pair.

6.3.2 Claim Extraction Results

Table 6.3.5 summarizes the claim extraction statistics per model.

Model	Inst.	Raw	Final	Avg/Inst	Grounding	Avg Score
Llama 3.1 70B	18	35	30	1.67	85.7%	4.83
Qwen2.5-72B	18	34	28	1.56	82.4%	5.00
Qwen2.5-Coder-32B	18	24	21	1.17	87.5%	5.29
Qwen3 32B	18	27	24	1.33	92.3%	5.46

Table 6.3.5: Claim extraction statistics per model.

All models extract between 1.17 and 1.67 claims per instance on average, with grounding rates above 82%. Most claims achieve *weak_file* grounding, indicating that extracted behaviors are generally linked to files modified by the patch.

6.3.3 Agentic Test Generation Results

Model Interaction Analysis

Figure 6.3.7 presents the discriminative rate for each combination of claim extraction and test generation models.

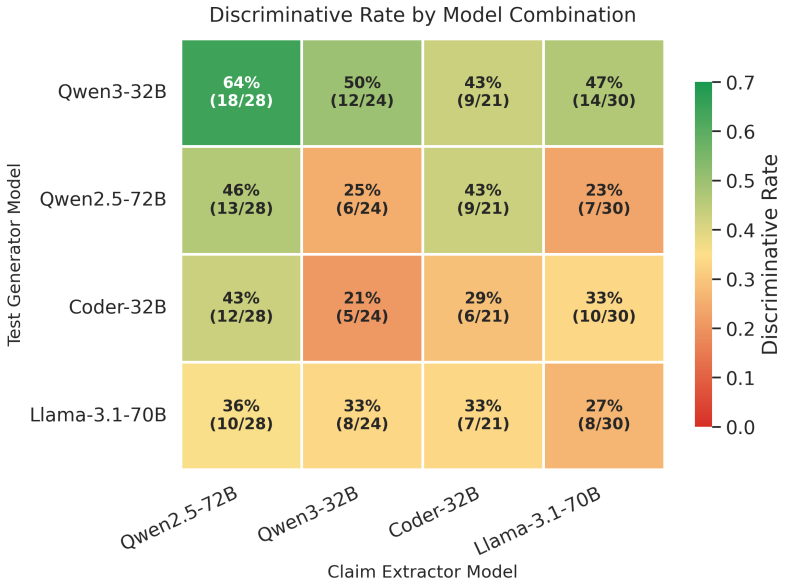

Figure 6.3.7: Discriminative rate across model combinations (test generator vs claim extractor).

The results reveal a strong interaction effect between claim extraction and test generation. Qwen3-32B consistently performs best as a test generator, particularly when paired with Qwen2.5-72B for claim extraction, achieving the highest discriminative rate (64%). With 18/28 claim-tests labeled as discriminative.

Cross-model combinations generally outperform same-model pairings, suggesting that diversity between extraction and generation reduces shared failure modes.

Instance Coverage

Across the 18 evaluated instances, 11 (61%) achieve at least one discriminative test across all configurations. Under the best-performing model combination, 9 instances are covered with a 100% success rate for the evaluated claims.

The remaining instances correspond to cases where no actionable claims were extracted or where test generation consistently failed due to environmental or behavioral complexity.

Overall Outcome Distribution

Figure 6.3.8 shows the distribution of outcome classifications across repositories and claim types from all the model combinations.

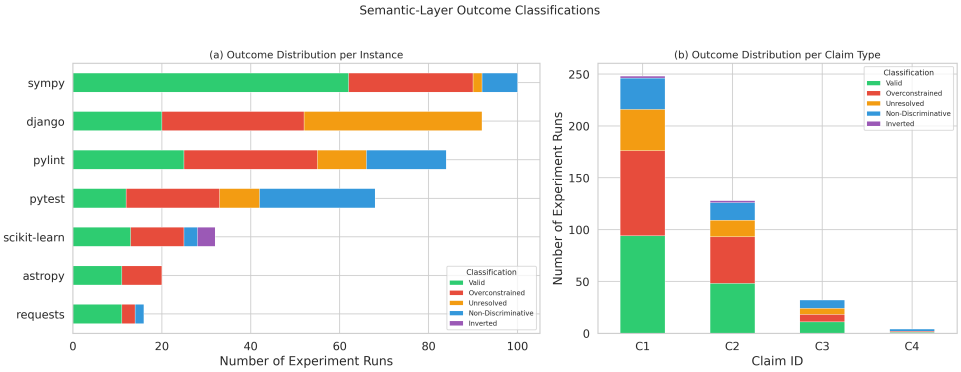

Figure 6.3.8: Outcome distribution of the agentic loop across repositories and claim types.

Across all 412 runs, 154 (37.4%) produce discriminative tests (95% CI: [32.8%, 42.2%]). The most common failure mode is *overconstrained* (135 runs, 32.8%), followed by *unresolved* (62, 15.0%) and *non-discriminative* (57, 13.8%). Only 4 runs (1.0%) yield *inverted* results. The dominance of overconstrained outcomes indicates that the agent tends to generate overly strict assertions rather than insufficient ones, a safer failure mode for verification, since it errs on the side of rejecting correct patches rather than accepting incorrect ones.

Focusing on the best model combination (Qwen3-32B as test generator / Qwen2.5-72B as claim extractor), the discriminative rate rises to 18 out of 28 runs (64.3%, 95% CI: [46.7%, 78.8%]), a 26.9 percentage-point improvement over the aggregate 37.4%. As shown in Figure 6.3.9 (a),

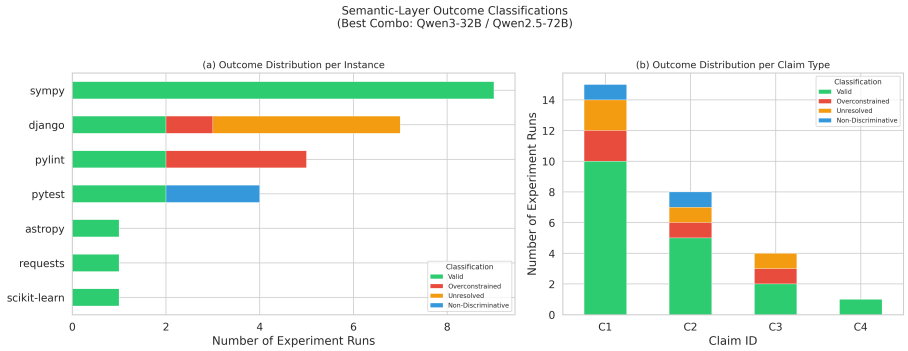

Figure 6.3.9: Outcome distribution of the agentic closed loop for Qwen3-32B Test Generator/ Qwen2.5-72B Claim Extractor

the *sympy* project achieves a perfect 9/9 discriminative rate across all its claims, while *astropy*, *requests*, and *scikit-learn* also achieve full validity on their respective claims. The main source of failure is the *django* project, where 4 out of 7 runs remain unresolved, likely due to the complexity of Django's test infrastructure and middleware setup. The *pylint* project contributes 3 overconstrained outcomes, consistent with the difficulty of testing linter configuration edge cases.

Figure 6.3.9 (b) shows the outcome distribution per claim ordinal. Primary claims (C1) account for 10 of 15 valid results, confirming that the first extracted claim, typically the most directly tied to the bug description, is the easiest to verify. Higher-order claims (C2–C4) maintain a reasonable success rate (8/13 valid), suggesting that the claim extraction captures genuinely distinct behavioral aspects of each bug rather than redundant reformulations.

Convergence Behavior

Figure 6.3.10 illustrates the convergence behavior of the agentic loop.

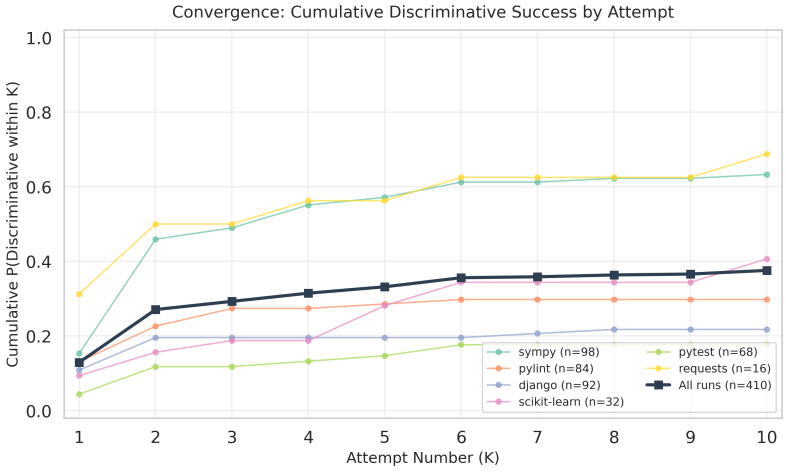

Figure 6.3.10: Cumulative discriminative success as a function of attempt number.

Most discriminative outcomes are reached within the first two attempts, with diminishing returns beyond attempt five. The per-project curves reveal substantial variation: `sympy` and `requests` plateau above 60%, while `django` and `pylint` remain below 30%, suggesting that convergence speed depends more on the complexity of the project's test infrastructure than on the number of iterations allowed.

6.3.4 Candidate Patch Validation

The candidate patch validation experiment applies the discriminative tests generated by the agentic loop to patches produced by LLM-based SWE-bench solvers, rather than the gold reference patch. This experiment tests the central hypothesis of the semantic layer: that discriminative claim tests can detect semantic divergences between patches that pass the same functional test suite.

For each discriminative instance/claim pair, the system collects all candidate patches from SWE-bench Lite experiments, both *resolved* (passing SWE-bench tests) and *failed* (not passing), and executes the discriminative test against each candidate in the same containerized environment used during the agentic loop. The candidate patch replaces the gold patch in the GOLD environment, while the BUG environment remains unchanged.

Reinterpretation of Classification Labels

In the candidate validation context, the BUG/GOLD classification scheme acquires a different interpretation than in the agentic loop. Since the discriminative test has already been validated against the gold patch (BUG=FAIL, GOLD=PASS), the outcome now reflects the *behavioral relationship between the candidate and the gold patch*:

- **ALIGNED** (BUG=FAIL, CANDIDATE=PASS): the candidate reproduces the gold patch's behavior on this claim.
- **DIVERGENT** (BUG=FAIL, CANDIDATE=FAIL): the candidate does not reproduce the gold patch's behavior despite passing the SWE-bench test suite, indicating a potential behavioral difference.
- **NON_DISCRIMINATIVE** (BUG=PASS, CANDIDATE=PASS): the test has lost its discriminative power in this execution context.
- **INVERTED** (BUG=PASS, CANDIDATE=FAIL): the candidate introduces the opposite behavior.
- **PATCH_FAIL**: the candidate patch could not be applied to the codebase.

We use the term *divergent* rather than *overconstrained* to distinguish this outcome from the agentic loop context, where overconstrained reflects test strictness rather than behavioral difference between implementations.

Scope and Filtering

The validation was executed using the best model combination identified in Section 6.3.3: Qwen3 32B for test generation and Qwen 72B for claim extraction, which produced 18 discriminative instance/claim pairs across 10 unique instances.

Of these, 13 pairs (8 instances) produced a *valid* gold classification during candidate validation, confirming that the discriminative test remains valid when re-executed in the candidate evaluation environment. The remaining 5 pairs, 4 from `sympy-17655` (gold classified as non-discriminative) and 1 from `pytest-5413` (gold classified as overconstrained), were excluded, as their tests lost discriminative power outside the original agentic loop environment[1]. All subsequent analysis is based on the 13 gold-valid pairs.

Overall Results

Table 6.3.6 presents the candidate patch validation results for all 13 gold-valid instance/claim pairs.

Instance	Clm.	RC	Resolved Classifications	FC	Failed Classifications
django-16255	C1	54	54 ALIGNED	0	—
django-16255	C2	54	54 ALIGNED	0	—
sympy-23262	C1	34	32 ALIGNED, 2 P.FAIL	17	16 DIVG., 1 ALIGNED
sympy-23262	C2	34	32 ALIGNED, 2 P.FAIL	17	16 DIVG., 1 ALIGNED
sympy-18057	C1	31	29 ALIGNED, 2 DIVG.	20	9 ALIGNED, 9 DIVG., 2 UNRES.
sympy-18057	C2	31	29 ALIGNED, 2 DIVG.	20	9 ALIGNED, 9 DIVG., 2 UNRES.
sympy-18057	C3	31	29 ALIGNED, 2 DIVG.	20	9 ALIGNED, 9 DIVG., 2 UNRES.
pytest-7168	C1	13	12 ALIGNED, 1 P.FAIL	40	21 DIVG., 18 ALIGNED, 1 P.FAIL
requests-2148	C1	8	8 ALIGNED	43	41 DIVG., 1 ALIGNED, 1 UNRES.
pylint-6506	C1	7	6 ALIGNED, 1 DIVG.	47	30 ALIGNED, 17 DIVG.
pylint-6506	C2	7	6 ALIGNED, 1 DIVG.	47	30 ALIGNED, 17 DIVG.
sk-learn-10508	C1	1	1 ALIGNED	52	50 ALIGNED, 2 DIVG.
astropy-7746	C1	0	—	50	42 ALIGNED, 7 DIVG., 1 UNRES.

Notes: RC = Resolved Candidates; FC = Failed Candidates; DIVG. = Divergent; P.FAIL = Patch Apply Failed.

Table 6.3.6: Candidate patch validation results using the best model combination (Qwen3 32B / Qwen 72B). Only gold-valid pairs are included.

[1]This is likely due to subtle differences in execution context between the agentic loop's Docker containers and the candidate validation environment, rather than a defect in the tests themselves.

Across the 13 gold-valid pairs, a total of 305 resolved patch evaluations were performed. Of these, 292 (95.7%) are classified as aligned (95% CI: [92.7%, 97.5%]), confirming behavioral equivalence with the gold patch. 8 evaluations (2.6%) are classified as *divergent*, and 5 (1.6%) failed due to patch application errors. On the failed-patch side, 373 evaluations were performed, with 164 (44.0%) classified as divergent and 200 (53.6%) as aligned.

Confidence intervals are computed using the Wilson score method, which provides more reliable estimates for proportions with small sample sizes.

Divergence Detection

Among 54 unique candidate agents evaluated, 3 (5.6%) were flagged as divergent despite passing the SWE-bench test suite:

1. ***sympy-18057 — isea* (Claude 3.5 Sonnet):** Divergent on all three claims (C1, C2, C3). This patch resolves the SWE-bench tests but does not reproduce the gold patch's behavior on the generated claim tests.

2. ***sympy-18057 — codefuse-cgm*:** Similarly divergent on all three claims. The consistency across claims strengthens the confidence that this represents a genuine behavioral difference rather than a test artifact.

3. ***pylint-6506 — Codev*:** Divergent on both claims (C1, C2). Among the 7 resolved patches for this instance, 6 are aligned while this one diverges, demonstrating that the claim test provides finer discrimination than the SWE-bench test suite.

Notably, for each flagged patch the divergent classification is consistent across *all* claims for that instance, providing cross-claim corroboration that the behavioral difference is genuine.

Case Study: Specification-Aligned Validation (requests-2148)

To illustrate the semantic layer operating as intended, consider `psf/requests-2148`. The issue reports that a `socket.error` raised during `iter_content` propagates uncaught instead of being wrapped as a `requests.exceptions.ConnectionError`. The claim extraction module produces a single claim, *"A socket.error should be caught and wrapped in a requests.exceptions.ConnectionError"*, which is a restatement of the user's report.

The agentic loop generates a test that mocks the raw socket to raise `socket.error` and asserts that `ConnectionError` is raised:

```
def test_claim_c1():
    r = Response()
    class RawMock:
        def stream(self, chunk_size, decode_content=None):
            raise socket.error("Simulated socket error")
    r.raw = RawMock()
    with pytest.raises(ConnectionError):
```

```
8           list(r.iter_content())
```

All 8 resolved candidates are classified as aligned, confirming that they implement the same catch-and-wrap behavior as the gold patch. Conversely, 41 of 43 failed candidates are divergent, the test correctly identifies that these patches do not handle the `socket.error`. This instance demonstrates that when the user's intent is specific and actionable, the semantic layer produces tests that are both faithful to the specification and strongly discriminative.

Case Study: Divergence Analysis (sympy-18057, pylint-6506)

To understand the nature of the detected divergences, we analyze the three flagged candidate patches.

Case 1: *sympy-18057*. Issue #18057 reports that comparing a SymPy Symbol with an arbitrary Python object via == triggers an AttributeError. The gold patch replaces sympify(other) with the strict _sympify(other), which raises SympifyError (caught by the existing handler), causing __eq__ to return False. Both flagged candidates instead restructure __eq__ to return NotImplemented for non-SymPy objects.

The SWE-bench test uses == and asserts (x == BadRepr()) is False, which passes for both approaches because Python's comparison protocol falls back to identity comparison when both sides return NotImplemented. However, the generated claim test calls __eq__ directly:

```
result = x.__eq__(c_instance)
assert result is False  # NotImplemented is not False
```

This assertion distinguishes the gold patch's False from the candidates' NotImplemented, a real implementation difference with implications for Python's comparison protocol. However, the user's issue exclusively uses the == operator, meaning the test is stricter than the stated behavioral intent.

Case 2: *pylint-6506*. Issue #6506 reports that passing an unrecognized option produces a traceback instead of a clean error message. The gold patch delegates to argparse.ArgumentParser.error(), outputting "Unrecognized option found: Q". The Codev candidate prints *"Error: Unrecognized option(s): Q"* manually.

The SWE-bench test checks "Unrecognized option" in output, a loose match both patches satisfy. The claim test is more specific:

```
assert "Unrecognized option found" in combined_output
```

The word "found" matches the gold patch's argparse output but not the Codev candidate's message. The verdict hinges on a single word in the error string.

Discussion. Both cases reveal the same pattern: the generated tests detect *real* implementation differences, but the assertions are calibrated against the gold patch's specific behavior rather than the user's abstract intent. The agentic loop never exposes the gold patch's source code to the test-generation model, the agent receives only the claim text and execution feedback. However, the iterative refinement creates a *selection pressure* toward gold-compatible assertions: when an initial loose assertion passes on both environments (non-discriminative), the feedback loop prompts the agent to tighten it until it fails on BUG and passes on GOLD. In doing so, the agent may converge on assertions specific to the gold patch's implementation choices rather than the user's minimal intent.

We refer to this phenomenon as *gold-patch anchoring*, a form of reference-induced bias arising in validation schemes where correctness is defined relative to a reference implementation: the generated tests implicitly encode the reference implementation's behavior through selection pressure, not through direct knowledge transfer. This explains why 95.7% of resolved candidates, which implement the same behavioral fix, are classified as aligned, while the few that diverge in implementation details are flagged.

This anchoring should not be interpreted as overfitting or leakage, but as an emergent property of the validation-driven optimization process. The divergent classification should therefore be read as "diverges from the gold patch's behavior" rather than "violates user intent."

This phenomenon is not unique to the proposed system, but rather reflects a well-established property of reference-based validation. In differential testing, regression test generation, and overfitting patch detection, test behavior is inherently anchored to a reference implementation, as correctness is defined relative to that reference. The semantic layer therefore exhibits a similar effect, with the key difference that test generation is guided by natural-language behavioral claims rather than purely programmatic transformations.

This connection reinforces that gold-patch anchoring should be understood as an instance of reference-induced bias in oracle-based validation frameworks.

Clean Instances

Several instances demonstrate strong agreement between SWE-bench resolution and semantic validation:

- *django-16255*: all 54 resolved patches classified as aligned across both claims, with zero failed candidates, indicating complete consensus among SWE-bench solvers.
- *requests-2148*: all 8 resolved patches aligned, while 41 of 43 failed patches are divergent.
- *sympy-23262*: 32 of 34 resolved patches aligned (2 patch application failures), while 16 of 17 failed patches are divergent.
- *pytest-7168*: 12 of 13 resolved patches aligned (1 patch application failure).

These results confirm that discriminative claim tests positively correlate with functional correctness: patches that genuinely fix the bug also satisfy the behavioral specification. The high alignment rate among resolved patches (95.7%) indicates that the tests are appropriately calibrated, strict enough to detect divergences but not so strict as to produce excessive false positives.

Failed Patches as Negative Controls

Across all 373 failed patch evaluations, the predominant classifications are divergent (44.0%) and aligned (53.6%). The distribution varies significantly by instance: for *requests-2148*, 95.3% of failed patches are divergent (the patch does not implement the catch-and-wrap behavior), while for *scikit-learn-10508*, 96.2% are aligned (the patch fixes the tested behavior despite failing the full SWE-bench suite for other reasons). This variance reflects the different granularity of the claims: some capture the core behavioral change, while others test an aspect that correlates less strongly with the complete test suite.

6.3.5 Interpretation and Limitations

It is important to distinguish between *reference-aligned correctness* and *specification-level correctness*. A test is classified as valid if it distinguishes between the buggy and reference implementations, but this does not necessarily imply correctness with respect to the natural-language specification.

The results reveal a phenomenon of *reference-induced anchoring*, consistent with behaviors observed in differential and regression-based testing frameworks. This effect emerges from the discriminative validation scheme (BUG=FAIL, GOLD=PASS), which defines correctness relative to a reference implementation and therefore introduces a bias toward reference-aligned assertions during iterative refinement.

As a result, the semantic layer should be interpreted as providing reference-based behavioral validation rather than fully specification-driven verification. While it introduces a stronger correctness signal than test-suite-based evaluation alone, it may reject alternative implementations that are semantically valid but differ from the reference.

6.3.6 Summary of Findings

The main findings of the semantic verification layer are:

- All four evaluated LLMs successfully extract grounded behavioral claims from issue descriptions, with grounding rates above 82%.
- The agentic loop produces discriminative tests in 37.4% of 412 runs, covering 11 of 18 instances (61.1%).
- The best model combination (Qwen3 32B / Qwen2.5 72B) achieves a 64.3% discriminative rate with 18/28 valid tests across 10 instances.

- Cross-model configurations outperform same-model pairings, suggesting that diversity between extraction and generation reduces shared failure modes.
- Most successful generations converge within two attempts (mean = 2.6, median = 2).
- Candidate patch validation on 305 resolved patches yields a 95.7% alignment rate, confirming that discriminative claim tests positively correlate with functional correctness.
- 3 unique candidate patches (5.6% of 54 resolved experiments) are flagged as divergent despite passing the SWE-bench test suite, demonstrating finer-grained behavioral discrimination than test suites alone.
- The dominant failure mode is overconstrained test generation (32.8%), indicating the agent produces overly strict assertions rather than insufficient ones.

Overall, the results demonstrate that claim-based semantic verification can capture meaningful behavioral differences beyond traditional test-suite evaluation. However, the generated tests exhibit *gold-patch anchoring*: through iterative refinement against the reference implementation, assertions implicitly encode the gold patch's behavior rather than the user's minimal specification. The semantic layer should therefore be interpreted as providing reference-aligned behavioral validation, a stronger signal than test suites alone, but not a fully specification-driven verification mechanism.

6.3.7 Threats to Validity

This section discusses potential threats to the validity of the experimental results.

Internal Validity. The semantic verification pipeline relies on LLM-generated claims and tests, which may introduce variability due to stochastic generation and prompt sensitivity. Although the agentic loop mitigates this through iterative refinement, some results may depend on model-specific behaviors or prompt configurations. Additionally, execution differences between environments (e.g., container setups) may affect reproducibility, as observed in a small number of cases where discriminative tests lost validity during candidate evaluation.

External Validity. The evaluation is conducted on a subset of 18 SWE-bench Lite instances due to execution environment constraints. While these instances span multiple repositories, they may not fully represent the diversity of real-world bugs or software systems. Consequently, the results should be interpreted as case-study evidence rather than statistically representative of the full benchmark or of general software engineering scenarios.

Construct Validity. The notion of correctness in the semantic layer is defined relative to a reference implementation. As a result, classifications such as ALIGNED and DIVERGENT reflect behavioral agreement with the gold patch rather than absolute correctness with respect to the natural-language specification. This introduces a dependency on the reference implementation as an oracle, which may not capture all valid interpretations of the intended behavior.

Conclusion Validity. The reported metrics are based on a limited number of instances and runs, particularly for per-configuration evaluations (e.g., 28 runs for the best model combination). Although confidence intervals are provided to account for statistical uncertainty, the relatively small sample sizes may still lead to variability in estimated performance. Therefore, conclusions regarding model comparisons and discriminative rates should be interpreted with appropriate caution.

6.3.8 End-to-End Verification Performance

This section synthesizes the results across all verification layers to evaluate the effectiveness of the proposed multi-layer verification harness as a unified system.

Table 6.3.7 summarizes the contribution of each layer in terms of coverage and verification signal.

Layer	**Instances**	**Pass Rate**	**Verification Signal**
Static	300	100% syntax valid, 87.7% Good+ SQI	Structural quality
Dynamic	18	17/18 (94.4%) resolved	Test-suite alignment
Semantic	18	Up to 64.3% discriminative (best config)	Reference-aligned behavior

Table 6.3.7: Cross-layer summary of the verification pipeline. Each layer provides a complementary signal, progressively increasing verification specificity.

The three layers provide progressively stronger but more constrained verification signals. The static layer offers full coverage and efficiently filters structurally invalid or low-quality patches. The dynamic layer evaluates whether patches satisfy the existing test suite, providing a functional validation signal relative to the benchmark. The semantic layer introduces an additional behavioral signal by generating discriminative tests derived from issue descriptions.

However, this progression comes at the cost of reduced coverage. While the static layer is evaluated on all 300 instances, the dynamic and semantic layers operate on a subset of 18 instances due to execution environment constraints. Consequently, end-to-end results should be interpreted as case-study evidence rather than statistically representative estimates of the full dataset.

The semantic layer further restricts coverage: only 11 of the 18 instances produce at least one discriminative test across all model configurations, and the best single configuration covers 10 instances. This reflects both the difficulty of extracting actionable claims and the challenges of generating executable tests in complex environments.

Importantly, the verification signals across layers are complementary but not equivalent. Passing the dynamic layer indicates that a patch satisfies the existing test suite, but does not guarantee alignment with the intended behavior described in the issue. The semantic layer addresses this gap

by introducing claim-based tests, which can expose behavioral differences that are not captured by the original tests.

This complementary behavior is illustrated by instances such as `pylint-6506`, where multiple candidate patches pass the SWE-bench test suite but are classified as DIVERGENT by the semantic layer, and `requests-2148`, where the semantic test faithfully captures the user's specification and cleanly separates resolved from failed patches.

At the same time, the semantic signal must be interpreted carefully. An ALIGNED classification indicates behavioral equivalence with the reference implementation, rather than an absolute notion of correctness. As discussed in Section 6.3.4, the generated tests exhibit *gold-patch anchoring*: through iterative refinement, assertions implicitly encode the gold patch's behavior rather than the user's minimal specification.

Overall, the results provide partial support for the central hypothesis. The multi-layer verification harness demonstrates that combining static, dynamic, and claim-based semantic analysis yields a richer and more informative evaluation signal than test-suite-based validation alone.

However, the system should be understood as performing *reference-based behavioral validation* rather than fully specification-driven verification. While the semantic layer can detect meaningful behavioral divergences, 3 out of 54 resolved experiments flagged as divergent despite passing the SWE-bench test suite, its conclusions remain conditioned on the reference implementation used during the discriminative testing process.

Chapter 7

Discussion and Future Work

7.1 Discussion of Results

The results demonstrate that combining static, dynamic, and semantic verification provides a more comprehensive evaluation framework than test-suite-based validation alone. While the dynamic layer confirms whether a patch satisfies existing tests, the semantic layer introduces behavioral checks derived from issue descriptions, enabling the detection of semantic differences that are not captured by the original test suite.

However, the semantic layer should not be interpreted as providing absolute correctness guarantees. As discussed in Section 6.3.5, the validation process is inherently reference-based, meaning that correctness is evaluated relative to the gold patch rather than the full natural-language specification. This distinction is critical for interpreting the results and understanding the scope of the proposed approach.

This highlights the central contribution of the proposed approach: the ability to operationalize natural-language specifications into executable tests that provide an additional verification signal beyond existing test suites.

7.2 Limitations

Several limitations of the proposed system should be acknowledged.

First, the semantic layer depends on the availability of executable environments, restricting evaluation to a subset of instances. This limits the scalability of the approach in its current form.

Second, the reliance on a reference implementation introduces a bias toward reference-aligned behavior, which may penalize valid alternative solutions. This phenomenon, referred to as gold-

patch anchoring, reflects a fundamental limitation of reference-based validation.

Third, the quality of the generated tests depends on both the effectiveness of claim extraction and the quality of the input problem description. The semantic layer relies on natural-language issue reports to derive behavioral claims, and therefore assumes that the user-provided description is sufficiently precise and actionable. In practice, issue descriptions may be incomplete, ambiguous, or underspecified, leading to incorrect or overly generic claims. This ambiguity propagates through the pipeline, affecting test generation and potentially resulting in tests that do not accurately capture the intended behavior.

Finally, the agentic loop, while effective, introduces additional computational cost due to iterative refinement and repeated execution.

7.3 Future Work

Several directions can be explored to extend this work.

A primary avenue is reducing the dependence on reference implementations. Future systems could incorporate multiple candidate patches or use natural-language entailment techniques to approximate specification-level validation.

Another direction is improving the robustness of claim extraction. Incorporating structured issue representations or leveraging additional context, such as commit history or documentation, could lead to more accurate behavioral claims.

From a systems perspective, expanding the availability of execution environments would enable evaluation at a larger scale. Techniques such as automated container synthesis or lightweight sandboxing could improve coverage.

From a practical perspective, the current prototype includes a user-facing interface that enables inspection of claims, generated tests, and verification outcomes. Future work may extend this interface into a more robust and scalable system, supporting larger-scale evaluation, improved visualization of verification signals, and integration into development workflows. In particular, incorporating interactive debugging features, richer visual analytics, and tighter coupling with code repositories could facilitate real-world adoption of the proposed verification framework.

This direction is particularly relevant for the semantic layer, where interpretability of generated claims and tests is essential. A richer implementation could support human-in-the-loop validation, enabling users to assess whether generated tests capture the intended behavior or reflect reference-induced biases.

Finally, integrating the semantic layer into the patch generation process could enable feedback-driven repair systems, in which discriminative tests guide the synthesis of higher-quality patches.

Chapter 8

Conclusions

This thesis presented a multi-layer verification harness for evaluating LLM-generated code patches, combining static analysis, dynamic execution, and claim-based semantic verification.

The results demonstrate that traditional test-suite-based evaluation, while effective for assessing functional correctness, is insufficient to capture finer-grained behavioral differences between implementations. The proposed semantic layer addresses this limitation by generating discriminative tests derived from natural-language issue descriptions, enabling the detection of divergences that remain undetected by existing test suites.

A key contribution of this work is the operationalization of natural-language specifications into executable behavioral claims, which can be used to systematically evaluate patch behavior. The experimental results show that this approach provides an additional verification signal that complements existing evaluation methods, particularly in identifying semantically divergent patches that pass standard benchmarks.

At the same time, the study highlights an important limitation: semantic verification operates within a reference-based validation framework, where correctness is defined relative to a gold patch. This leads to the phenomenon of gold-patch anchoring, which reflects a broader property of oracle-based evaluation rather than a limitation specific to the proposed system. As a result, the semantic layer should be interpreted as providing reference-aligned behavioral validation, rather than fully specification-driven verification.

Overall, the findings demonstrate that combining multiple verification signals yields a more comprehensive evaluation framework for LLM-based code generation systems. While challenges remain in achieving fully specification-level validation, the proposed approach represents a step toward more reliable and interpretable evaluation of automatically generated code patches.

Appendix A

Static Layer Implementation Details

This appendix presents the implementation details of the static verification layer described in Section 5.3. The two sections below cover the syntax validation and structural analysis pipeline and the computation of the Static Quality Index.

A.1 Syntax Validation and Structural Analysis

The syntax and structural analysis module operates on each Python file modified by the candidate patch. It proceeds through four stages: diff scoping, syntax validation via AST parsing, structural metric extraction, and changed-function identification.

A.1.1 Patch Scoping

Modified files and their changed line ranges are extracted from the unified diff using a lightweight parser. The parser identifies file paths from `+++ b/` headers and extracts hunk ranges from `@@` headers using the regular expression shown in Listing A.1.

```
HUNK_RE = re.compile(r"@@ -\d+(?:,\d+)? \+(\d+)(?:,(\d+))? @@")

def parse_unified_diff(diff_text: str) -> Dict[str, List[Tuple[int, int
    ]]]:
    result = {}
    current_file = None
    for line in diff_text.splitlines():
        if line.startswith("+++ "):
            path = line[4:].strip()
            if path.startswith("b/"):
                path = path[2:]
            current_file = path
            result.setdefault(current_file, [])
        elif line.startswith("@@"):
            match = HUNK_RE.search(line)
```

```
            if match and current_file:
                start = int(match.group(1))
                length = int(match.group(2) or "1")
                result[current_file].append((start, start + length - 1))
    return result
```

Listing A.1: Hunk header regular expression and diff parser

Only Python files are retained for subsequent analysis using a simple extension-based filter:

```
def filter_paths_to_py(paths: List[str]) -> List[str]:
    return [p for p in paths if p.endswith(".py")]
```

Listing A.2: Python file filter

A.1.2 Syntax Validation and Structural Metrics

Each modified Python file is parsed using the `ast` module from the Python standard library. A successful parse confirms that the patched file is syntactically valid. From the resulting AST, four structural metrics are extracted: the number of function definitions, the number of class definitions, the maximum nesting depth, and the average function length in lines of code.

```
def get_ast_depth(node, current=0):
    if not list(ast.iter_child_nodes(node)):
        return current
    return max(
        get_ast_depth(child, current + 1)
        for child in ast.iter_child_nodes(node)
    )
```

Listing A.3: AST nesting depth computation

```
def syntax_ast_validation(file_path: Path) -> dict:
    try:
        source = Path(file_path).read_text(encoding="utf-8")
        tree = ast.parse(source)

        n_funcs = sum(isinstance(n, ast.FunctionDef) for n in ast.walk(tree))
        n_classes = sum(isinstance(n, ast.ClassDef) for n in ast.walk(tree))
        ast_depth = get_ast_depth(tree)

        func_lengths = []
        for node in ast.walk(tree):
            if isinstance(node, ast.FunctionDef):
                if hasattr(node, "end_lineno") and hasattr(node, "lineno"):
                    func_lengths.append(node.end_lineno - node.lineno)
```

```
        avg_func_length = (
            sum(func_lengths) / len(func_lengths) if func_lengths else
                0
        )

        return {
            "is_code_valid": True,
            "n_functions": n_funcs,
            "n_classes": n_classes,
            "ast_depth": ast_depth,
            "avg_func_length": avg_func_length,
            "error": None,
        }
    except SyntaxError as e:
        return {
            "is_code_valid": False,
            "error": f"{type(e).__name__} at line {e.lineno}: {e.msg}",
        }
```

Listing A.4: Syntax validation and structural metric extraction

A.1.3 Changed Function Extraction and AST Diff Ratio

Once the structural metrics are available, the system identifies which functions are directly affected by the patch. This is done by checking whether the line range of each function definition in the AST overlaps with any of the modified line ranges extracted from the diff.

```
def extract_changed_functions(
    file_path: Path,
    diff_ranges: List[Tuple[int, int]]
) -> List[str]:
    changed_funcs = []
    with open(file_path, "r", encoding="utf-8") as f:
        tree = ast.parse(f.read())

    for node in ast.walk(tree):
        if isinstance(node, ast.FunctionDef):
            start = node.lineno
            end = getattr(node, "end_lineno", start)
            for (diff_start, diff_end) in diff_ranges:
                if not (end < diff_start or start > diff_end):
                    changed_funcs.append(node.name)
                    break
    return changed_funcs
```

Listing A.5: Changed function extraction via line-range overlap

The AST diff ratio is then computed to quantify the structural breadth of the modification:

```
def compute_ast_diff_ratio(n_functions, n_classes, changed_functions):
```

```
    total_entities = n_functions + n_classes
    if total_entities == 0:
        return 0.0
    return min(len(changed_functions) / total_entities, 1.0)
```

Listing A.6: AST diff ratio computation

A.2 Static Quality Index Computation

The Static Quality Index (SQI) aggregates the results of five static analysis tools into a single score in the range $[0, 100]$. Each tool's raw output is first normalized to a common $[0, 100]$ scale, then combined through a weighted sum.

A.2.1 Default Configuration

Table A.2.1 summarizes the default weights and the enabled analyzers.

Analyzer	Weight	Evaluated Aspect
Pylint	0.50	Code correctness and conventions
Radon MI	0.25	Maintainability index
Flake8	0.15	PEP 8 style compliance
Mypy	0.05	Type consistency
Bandit	0.05	Security vulnerabilities

Table A.2.1: Default SQI analyzer weights

When a subset of analyzers is disabled, the active weights are renormalized so that they sum to 1:

$$w'_k = \frac{w_k}{\sum_{j \in \mathcal{A}} w_j}, \quad k \in \mathcal{A} \tag{A.1}$$

where $\mathcal{A}$ denotes the set of active analyzers.

A.2.2 Per-Tool Normalization

Each analyzer's output is mapped to a $[0, 100]$ score as follows.

Pylint. Pylint produces a score on a $[0, 10]$ scale. The normalization is a simple linear rescaling:

$$S_{\text{pylint}} = \frac{\text{score}_{\text{pylint}}}{10} \times 100 \tag{A.2}$$

Radon Maintainability Index. The Radon Maintainability Index is already expressed on a $[0, 100]$ scale and is used directly after clamping:

$$S_{\text{radon}} = \text{clamp}(\text{MI},\ 0,\ 100) \tag{A.3}$$

Flake8. Flake8 reports individual style violations, each identified by a code whose first character indicates its category. Different categories carry different severity weights, as shown in Table A.2.2.

Code Prefix	Weight (w_i)	Category
F	3.0	PyFlakes (logical errors)
E	1.0	PEP 8 errors
W	0.5	PEP 8 warnings
C	0.8	Complexity / McCabe
N	0.8	Naming conventions
D	0.8	Docstring conventions

Table A.2.2: Flake8 violation category weights

The weighted violations are aggregated into a penalty relative to the file size:

$$S_{\text{flake8}} = \max\left(0,\ \left(1 - \frac{\sum_i w_i}{\text{LOC} \times 0.5}\right) \times 100\right) \tag{A.4}$$

where the denominator $\text{LOC} \times 0.5$ acts as a tolerance threshold proportional to the size of the file.

Mypy. Type errors detected by Mypy are normalized using a tolerance constant $\gamma = 50$:

$$S_{\text{mypy}} = \max\left(0,\ \left(1 - \frac{n_{\text{errors}}}{\gamma + \text{LOC}}\right) \times 100\right) \tag{A.5}$$

The addition of γ in the denominator prevents the score from dropping sharply for small files with few type annotations.

Bandit. Security issues reported by Bandit are weighted by severity using the coefficients in Table A.2.3 and normalized against a tolerance constant $\beta = 10$:

$$S_{\text{bandit}} = \max\left(0,\ \left(1 - \frac{\sum_s w_B(s) \cdot n_s}{\beta}\right) \times 100\right) \tag{A.6}$$

where n_s is the number of issues at severity level s.

Severity	Weight (w_B)
High	5
Medium	3
Low	1

Table A.2.3: Bandit severity weights

A.2.3 Weighted Aggregation

The final SQI is computed as the weighted sum of the normalized scores:

$$\text{SQI} = \sum_{k \in \mathcal{A}} w'_k \cdot S_k \tag{A.7}$$

Listing A.7 shows the implementation of the full computation.

```
def compute_sqi(pylint_score, radon_mi, flake8_issues,
                mypy_errors, bandit_counts, loc,
                weights=DEFAULT_WEIGHTS, checks=DEFAULT_CHECKS):

    # Renormalize weights to active checks
    active = {k: v for k, v in weights.items() if checks.get(k, True)}
    total = sum(active.values())
    active = {k: v / total for k, v in active.items()}

    # --- Normalize each sub-metric to [0, 100] ---
    pylint_norm = (pylint_score / 10) * 100

    radon_norm = max(0.0, min(100.0, radon_mi))

    # Flake8: weighted penalty by violation category
    w_f8 = {"F": 3.0, "E": 1.0, "W": 0.5, "C": 0.8, "N": 0.8, "D": 0.8}
    weighted_sum = sum(w_f8.get(i["code"][0], 1.0) for i in
        flake8_issues)
    penalty = min(1.0, weighted_sum / (loc * 0.5))
    flake8_norm = max(0.0, (1 - penalty) * 100)

    # Mypy: tolerance constant gamma = 50
    gamma = 50
    mypy_norm = max(0.0, (1 - mypy_errors / (gamma + loc)) * 100)

    # Bandit: severity-weighted with tolerance beta = 10
    beta = 10
    w_B = {"HIGH": 5, "MEDIUM": 3, "LOW": 1}
    weighted_bandit = sum(w_B[k] * v for k, v in bandit_counts.items())
    bandit_norm = max(0.0, (1 - weighted_bandit / beta) * 100)

    # --- Weighted aggregation ---
    scores = {"pylint": pylint_norm, "radon": radon_norm,
```

```
                "flake8": flake8_norm, "mypy": mypy_norm,
                "bandit": bandit_norm}
    sqi = sum(active[k] * scores[k] for k in active)

    # --- Classification ---
    if sqi >= 85:   label = "Excellent"
    elif sqi >= 70: label = "Good"
    elif sqi >= 50: label = "Fair"
    else:           label = "Poor"

    return {"SQI": round(sqi, 2), "classification": label}
```

Listing A.7: Static Quality Index computation

Appendix B

Dynamic Layer Implementation Details

This appendix documents the implementation details of the dynamic verification layer introduced in Section 5.4. The main chapter presents the methodology and design decisions, while this appendix provides implementation details required for reproducibility.

B.1 Environment Reproduction Pipeline

The environment reproduction pipeline converts a SWE-bench instance identifier into a runnable execution environment. The process consists of three stages:

1. Repository reconstruction and patch application
2. Container image resolution
3. Docker-to-Singularity conversion

B.1.1 Repository Loading and Patch Application

Repository cloning and patch application are performed through a lightweight loader component that reconstructs the exact repository state defined by the SWE-bench instance metadata.

```
# Phase 1: Shallow clone
git clone --depth 1 https://github.com/<org>/<repo>.git <workdir>

# Phase 2: Checkout base commit (two-stage strategy)
git fetch --depth 1 origin <base_commit>    # try shallow first
git fetch --unshallow                       # fallback if needed
git checkout <base_commit>

# Phase 3: Apply model patch
git apply --whitespace=fix model.patch
```

```

# Phase 4: Apply test patch (if present in SWE-bench metadata)
git apply --whitespace=fix test.patch
```

Listing B.1: Repository cloning and patch application sequence.

Each instance receives an isolated working directory derived from the repository name and task identifier (e.g., *repos_temp_42/django__django/*). The *–whitespace=fix* flag tolerates minor whitespace inconsistencies in model-generated patches, which are common when patches are produced by language models that may not preserve exact indentation conventions.

B.1.2 Docker Image Resolution

SWE-bench instances rely on containerized environments distributed as Docker images. The system maps instance identifiers of the form *¡org¿__¡repo¿-¡version¿* to candidate image names across several registries.

The resolver attempts the following naming patterns in priority order:

1. *swebench/sweb.eval.x86_64.*$< org >$*_1776_*$< repo > - < version >$*:latest*
2. *ghcr.io/swe-bench/sweb.eval.x86_64.*$< org >$*_1776_*$< repo > - < version >$*:latest*
3. *aorwall/swe-bench-*$< repo >:< instance\_id >$
4. *swebench/*$< repo >:< instance\_id >$

The first two correspond to official SWE-bench images, while the latter patterns serve as fallbacks for community-maintained registries.

B.1.3 Docker-to-Singularity Conversion

Because Docker cannot run on the target HPC cluster, container images are converted to the Singularity Image Format (*.sif*). Two conversion paths are supported depending on whether the Docker daemon is available.

```
# Path A: Docker daemon available (e.g., login node)
docker pull <image>
singularity build --fakeroot output.sif docker-daemon://<image>

# Path B: No Docker daemon (e.g., compute node)
export APPTAINER_DOCKER_USERNAME=<user>
export APPTAINER_DOCKER_PASSWORD=<token>
singularity build --fakeroot output.sif docker://<image>
```

Listing B.2: Singularity build commands for both conversion paths.

Path A leverages Docker's native authentication mechanism, which tends to be more reliable for pulling from registries with rate limits. **Path B** is used on compute nodes where Docker is unavailable and requires explicit credentials via environment variables. The *–fakeroot* flag allows the build to proceed without root privileges by using user namespace remapping.

The build process retries up to three times with exponential backoff (delays of 5, 10, and 15 seconds) to handle transient failures. On success, the resulting *.sif* file is moved to the cache directory under `<cache_root>/<repo_name>/<instance_id>.sif`, where it is available for reuse by subsequent tasks.

B.2 Container Execution and Path Configuration

When tests run inside the container, Python packages may originate from three sources:

- patched repository source code
- precompiled extensions from the container image
- auxiliary packages installed during evaluation

The *PYTHONPATH* ordering ensures that patched repository code always takes precedence over installed dependencies:

```
# 1. src-layout packages (e.g., pytest uses src/pytest/)
/workspace/src
# 2. Root-layout packages (e.g., django/, sympy/)
/workspace
# 3. lib-layout packages (e.g., matplotlib uses lib/matplotlib/)
/workspace/lib
```

Listing B.3: PYTHONPATH construction with precedence ordering.

B.2.1 Pytest Test Execution

For repositories using pytest, the full container invocation is:

```
singularity exec \
  --bind <repo_path>:/workspace \
  --pwd /workspace \
  --env "PYTHONPATH=/workspace/src:/workspace:/workspace/lib:/workspace
     /.pip_packages" \
  <image.sif> \
  /opt/miniconda3/envs/testbed/bin/python -m pytest \
  <FAIL_TO_PASS_tests> <PASS_TO_PASS_tests> -v --tb=short
```

Listing B.4: Singularity command for pytest-based test execution.

The *–bind* flag mounts the host repository into the container. The *–pwd* flag sets the working directory. The Python interpreter from the *testbed* conda environment is used explicitly to ensure the correct version and pre-installed dependencies.

B.2.2 Django Test Execution

Django repositories use a custom test runner (*tests/runtests.py*) rather than pytest. The system converts Django-style test identifiers from *test_name (module.Class)* to the dot-separated format *module.Class.test_name* expected by the runner:

```
singularity exec \
  --bind <repo_path>:/workspace \
  --pwd /workspace \
  --env "PYTHONPATH=/workspace:/workspace/.pip_packages" \
  --env "DJANGO_SETTINGS_MODULE=tests.test_sqlite" \
  <image.sif> \
  /opt/miniconda3/envs/testbed/bin/python \
  tests/runtests.py <converted_test_ids>
```

Listing B.5: Singularity command for Django test execution.

B.3 SLURM Configuration

Listing B.6 shows a representative SLURM array job script used to evaluate a batch of SWE-bench instances. Each array task processes one instance, identified by *SLURM_ARRAY_TASK_ID*.

```
#!/bin/pythonstyle
#SBATCH --job-name=swebench_eval
#SBATCH --account=cml-scavenger
#SBATCH --partition=cml-scavenger
#SBATCH --time=02:00:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --mem=8G
#SBATCH --array=1-500
#SBATCH --output=logs/eval_%A_%a.out

TASK_ID=${SLURM_ARRAY_TASK_ID}

# Per-task isolation
export REPOS_ROOT="repos_temp_${TASK_ID}"

python scripts/slurm/slurm_worker_integrated.py \
  --task-id ${TASK_ID} \
  --repos-root ${REPOS_ROOT} \
  --output-dir results/
```

Listing B.6: SLURM array job script for batch evaluation.

The *cml-scavenger* partition uses opportunistic scheduling, filling idle capacity across the cluster. The two-hour time limit accommodates the majority of instances; tasks that exceed this limit, typically due to large test suites or slow builds, are terminated by SLURM and recorded as timeouts.

B.4 Known Environment Workarounds

The bind-mount approach, running patched source from */workspace* rather than the container's built-in */testbed*, works seamlessly for most repositories. However, several projects require additional handling due to generated files, data dependencies, or version detection mechanisms that are not captured by copying C extensions alone. Table B.4.1 documents these cases.

Table B.4.1: Project-specific environment workarounds required for containerized execution.

Project	**Issue**	**Workaround**
Astropy	ERFA bindings (*_erfa/core.py*) and *version.py* are generated at build time by Cython and *setuptools_scm*; not present in source.	Copy *core.py*, *version.py*, and *cython_version.py* from */testbed/astropy/* to */workspace/astropy/*.
Matplotlib	Runtime data directory (*mpl-data/*) containing stylesheets, fonts, and colormaps is installed alongside the package, not in source. *_version.py* is also generated.	Copy *mpl-data/* directory and *_version.py* from */testbed/lib/matplotlib/* to */workspace/lib/matplotlib/*.
Pytest	*_pytest/_version.py* is generated by *setuptools_scm*. Without it, *pytest –version* reports *"unknown"*, which fails *minversion* checks in config files.	Copy *_version.py* from */testbed/src/_pytest/*. Additionally, comment out *minversion* directives in *tox.ini*, *pytest.ini*, and *setup.cfg*.

These workarounds are applied automatically during the package preparation phase (Section B.2.1) based on filesystem heuristics, for example, detecting Astropy by the presence of an *astropy/_erfa/* directory. While pragmatic, this approach requires maintenance as new repositories are added to the evaluation set. A more robust alternative would be to run tests entirely within */testbed* and overlay only the patched files, though this would complicate the bind-mount strategy for repositories with complex build systems.

Appendix C

Semantic Verification Layer: Implementation Details

This appendix provides implementation-level details for the semantic verification layer introduced in Section 5.5. While the main chapter presents the semantic layer at the architectural and methodological level, the appendix documents the concrete heuristics, data structures, scoring functions, and control-flow logic used in the prototype implementation. Unless otherwise indicated, the code listings shown in this appendix are simplified excerpts intended to illustrate the main logic of each component rather than reproduce the full implementation verbatim.

The appendix is organized into five parts. First, it presents an example SWE-bench problem statement used as input to the semantic layer. Second, it details the claim extraction pipeline, including eligibility checking, stacktrace extraction, JSON parsing, evidence verification, and specificity scoring. Third, it describes the grounding mechanism used to connect extracted claims to concrete code symbols. Fourth, it documents the agentic test-generation loop, including guardrails, dynamic planning playbooks, anti-pattern detection, diagnostics, verification classification, and loop termination logic. Finally, it summarizes the default configuration parameters used in the experiments.

C.1 Example SWE-bench Problem Statement

Listing C.1 shows a representative problem statement from the SWE-bench dataset. The issue describes a failure when passing empty arrays to a WCS transformation in the Astropy library and serves as the running example for the claim decomposition shown in Table 5.5.3 in Section 5.5.1.

```
1 Issue when passing empty lists/arrays to WCS transformations
2 The following should not fail but instead should return empty lists/
     arrays:
3
4 In [1]: from astropy.wcs import WCS
```

```

In [2]: wcs = WCS('2MASS_h.fits')

In [3]: wcs.wcs_pix2world([], [], 0)
---------------------------------------------------------------------------

InconsistentAxisTypesError                Traceback (most recent call
    last)
<ipython-input-3-e2cc0e97941a> in <module>()
----> 1 wcs.wcs_pix2world([], [], 0)

~/Dropbox/Code/Astropy/astropy/astropy/wcs/wcs.py in wcs_pix2world(self
    , *args, **kwargs)
   1352         return self._array_converter(
   1353             lambda xy, o: self.wcs.p2s(xy, o)['world'],
-> 1354             'output', *args, **kwargs)

~/Dropbox/Code/Astropy/astropy/astropy/wcs/wcs.py in _array_converter(
    self, func, sky, ra_dec_order, *args)
   1267                     "a 1-D array for each axis, followed by an
       origin.")
   1268
-> 1269             return _return_list_of_arrays(axes, origin)

~/Dropbox/Code/Astropy/astropy/astropy/wcs/wcs.py in
    _return_list_of_arrays(axes, origin)
   1223             if ra_dec_order and sky == 'input':
   1224                 xy = self._denormalize_sky(xy)
-> 1225             output = func(xy, origin)

InconsistentAxisTypesError: ERROR 4 in wcsp2s()
ncoord and/or nelem inconsistent with the wcsprm.
```

Listing C.1: Example problem statement extracted from an issue report describing a failure when passing empty arrays to a WCS transformation.

C.2 Claim Extraction Pipeline

C.2.1 Eligibility Check

The eligibility check scores the issue text according to the presence of actionable signals. An instance is marked eligible when the cumulative score meets or exceeds a configurable threshold (default: 2). Not all eligible issues exhibit all signals listed in Table C.2.1; the scoring scheme is intentionally redundant so that different issue-writing styles can still satisfy the threshold.

```
def check_issue_eligibility(
    problem_statement: str, *, threshold: int = 2
) -> EligibilityResult:
    score, reasons = 0, []
```

Signal	Points	Detection Method
Exception keyword	+2	Case-sensitive match against a set of standard Python exception names (e.g., `ValueError`, `TypeError`, `KeyError`).
Stacktrace present	+2	Regex match for `File "...", line \d+` or the string `Traceback`.
Behavioral keyword	+1	Match against phrases such as `should return`, `must raise`, `incorrectly`, `fails to`.
Code references	+1	Presence of backtick-delimited identifiers.
Function references	+1	Identifiers followed by parentheses (e.g., `foo()`).
Code blocks	+1	Fenced code blocks (```) or interactive prompts (>>>).

Table C.2.1: Eligibility scoring signals and their point values.

```
    text = problem_statement

    # Exception keywords (+2)
    for kw in ELIGIBILITY_EXCEPTION_KEYWORDS:
        if kw in text:
            score += 2; reasons.append(f"exception_keyword:{kw}")
            break

    # Stacktrace (+2)
    if re.search(r'File ["\'].+?["\'], line \d+', text) or "Traceback"
       in text:
        score += 2; reasons.append("stacktrace_present")

    # Behavioral keywords (+1)
    lower = text.lower()
    for kw in ELIGIBILITY_BEHAVIOR_KEYWORDS:
        if kw in lower:
            score += 1; reasons.append(f"behavior_keyword:{kw}")
            break

    # Code references (+1)
    if "`" in text:
        score += 1; reasons.append("code_reference")

    # Function references (+1)
    if re.search(r"\w+\(", text):
        score += 1; reasons.append("function_reference")

    # Code blocks (+1)
    if "```" in text or ">>>" in text:
```

```
        score += 1; reasons.append("code_block")

    return EligibilityResult(
        eligible=score >= threshold, score=score, reasons=reasons
    )
```

Listing C.2: Eligibility check implementation

C.2.2 Stacktrace Extraction

Error tracebacks embedded in the issue text are parsed to extract file paths and function names, which are used during evidence verification to award additional confidence when a claim's target symbols appear in the reported stacktrace.

```
_STACKTRACE_RE = re.compile(
    r'File ["\']([^"\']+\.py)["\'],\s*line\s*(\d+)'
    r'(?:,\s*in\s+(\w+))?'
)

def extract_stacktrace_info(
    problem_statement: str,
) -> List[Dict]:
    entries = []
    for m in _STACKTRACE_RE.finditer(problem_statement):
        path = m.group(1)
        # Exclude site-packages and virtual environments
        if "site-packages" in path or "/venv/" in path:
            continue
        entries.append({
            "file": path,
            "line": int(m.group(2)),
            "function": m.group(3),  # may be None
        })
    return entries
```

Listing C.3: Stacktrace extraction regex and function

C.2.3 JSON Parsing with Repair

LLM output is parsed using a multi-strategy fallback pipeline that strips markdown fencing, attempts direct JSON parsing, and applies repair heuristics for common malformations.

```
def parse_json_array_with_fallbacks(
    text: str,
) -> Tuple[List[Dict[str, Any]], Dict[str, Any]]:
    attempts = []

    # Strip markdown fences
    cleaned = re.sub(r"```(?:json)?\s*\n?", "", text).strip()

```

```
    # Strategy 1: Direct parse
    try:
        result = json.loads(cleaned)
        if isinstance(result, list):
            return result, {"attempts": [{"method": "direct",
                                          "success": True}]}
    except json.JSONDecodeError as e:
        attempts.append({"method": "direct", "success": False,
                         "error": str(e)})

    # Strategy 2: Locate array bounds
    start, end = cleaned.find("["), cleaned.rfind("]")
    if start != -1 and end > start:
        fragment = cleaned[start : end + 1]
        try:
            result = json.loads(fragment)
            if isinstance(result, list):
                return result, {"attempts": attempts + [
                    {"method": "find_array", "success": True}]}
        except json.JSONDecodeError:
            pass

        # Strategy 3: Repair trailing commas
        repaired = re.sub(r",(\s*[}\]])", r"\1", fragment)
        try:
            result = json.loads(repaired)
            if isinstance(result, list):
                return result, {"attempts": attempts + [
                    {"method": "repaired", "success": True}]}
        except json.JSONDecodeError as e:
            attempts.append({"method": "repaired", "success": False,
                             "error": str(e)})

    return [], {"attempts": attempts}
```

Listing C.4: JSON array parsing with fallback repair

C.2.4 Evidence Verification and Scoring

Each claim undergoes evidence verification against the issue text, code context, and extracted stacktraces. The scoring formula accumulates points from four signal types, capped at a maximum of 3.

```
def verify_evidence_expanded(
    claim: Dict[str, Any],
    problem_statement: str,
    code_context: str,
    stacktrace_info: Optional[List[Dict]] = None,
) -> Dict[str, Any]:
    score, matches = 0, []
    spans = claim.get("evidence", {}).get("spans", [])
```

```
    ps_lower = problem_statement.lower()

    for span in spans:
        # +2: exact (case-insensitive) match in issue text
        if span.lower() in ps_lower:
            score += 2
            matches.append({"source": "issue", "type": "exact"})
        else:
            # +1: partial match (>= 80% token overlap)
            span_tokens = set(span.lower().split())
            ps_tokens = set(ps_lower.split())
            if span_tokens and len(span_tokens & ps_tokens) / len(
                span_tokens) >= 0.8:
                score += 1
                matches.append({"source": "issue", "type": "partial"})

        # +1: span appears in docstring/comment (max once)
        if span.lower() in code_context.lower():
            score = min(score + 1, 3)
            matches.append({"source": "code_context", "type": "
                docstring"})
            break  # award at most once

    # +1: target symbol in stacktrace (max once)
    if stacktrace_info:
        sym_variants = set()
        for sym in claim.get("target_symbols", []):
            sym_variants |= normalize_symbol(sym)
        for entry in stacktrace_info:
            if entry.get("function") and entry["function"] in
                sym_variants:
                score = min(score + 1, 3)
                matches.append({"source": "stacktrace", "type": "
                    function"})
                break

    return {"score": min(score, 3), "matches": matches,
            "verified": score > 0}
```

Listing C.5: Evidence verification and scoring

C.2.5 Composite Claim Scoring

The composite score aggregates grounding strength, evidence verification, observability, and self-assessed confidence. Claims scoring below the minimum threshold (default: 2) are excluded from test generation.

```
OBSERVABLE_KEYWORDS = {
    "return", "returns", "raise", "raises", "exception", "error",
    "equal", "equals", "output", "result", "value", "state",
    "attribute", "property", "contain", "contains", "produce",
```

Component	Points	Criterion
Grounding	+3 / +1 / +0	Strong / weak (file or ref) / none
Evidence	+0 to +3	From evidence verification (see above)
Observable	+1 / −1	Presence / absence of observable keyword in claim text
Confidence	+1 / +0 / −1	High / medium / low

Table C.2.2: Composite claim scoring components.

```
    "must be", "should be", "will be", "not raise", "not throw",
    "is none", "is not none", "true", "false", "empty",
}

def compute_claim_score_v2(
    claim: Dict[str, Any],
    evidence_result: Dict[str, Any],
) -> Dict[str, Any]:
    score = 0

    # Grounding component
    grounding_status = claim.get("grounding_status", "none")
    if grounding_status == "strong":
        score += 3
    elif grounding_status in ("weak_file", "weak_ref"):
        score += 1

    # Evidence component
    score += evidence_result.get("score", 0)

    # Observable component
    text_lower = claim.get("claim_text", "").lower()
    has_obs = any(kw in text_lower for kw in OBSERVABLE_KEYWORDS)
    score += 1 if has_obs else -1

    # Confidence component
    conf = claim.get("confidence", "medium")
    score += {"high": 1, "medium": 0, "low": -1}.get(conf, 0)

    return {"computed_score": score, "meets_threshold": score >= 2}
```

Listing C.6: Composite claim scoring

C.2.6 Specificity Check

A separate specificity check determines whether a claim is concrete enough for test generation. All five criteria must be satisfied.

```
VAGUE_PHRASES = {"properly", "correctly", "should work",
                 "as expected", "handle"}
```

```

def compute_claim_specificity(claim: Dict[str, Any]) -> Dict[str, Any]:
    symbols = claim.get("target_symbols", [])
    then_text = claim.get("then", "").lower()
    given_text = claim.get("given", "")
    when_text = claim.get("when", "")

    checks = {
        "has_symbols": bool(symbols) and all(len(s) > 1 for s in
            symbols),
        "has_observable": any(kw in then_text for kw in
            OBSERVABLE_KEYWORDS),
        "not_vague": not any(v in then_text for v in VAGUE_PHRASES),
        "concrete_given": len(given_text) > 10 and not given_text.
            startswith("a "),
        "concrete_when": len(when_text) > 10,
    }
    return {"is_specific": all(checks.values()), "checks": checks}
```

Listing C.7: Claim specificity check

C.3 Grounding Implementation

C.3.1 Diff Parsing and Symbol Extraction

The grounding mechanism parses the unified diff into structured line records and extracts symbols defined or assigned in added or removed lines using four regular-expression patterns. The extractor focuses on function definitions, class definitions, constant assignments, and instance-attribute assignments because these patterns cover the most common symbol types referenced in issue reports while keeping diff parsing lightweight and robust.

```
@dataclass(frozen=True)
class DiffLine:
    path: str
    kind: Literal["add", "del", "ctx"]
    old_lineno: Optional[int]
    new_lineno: Optional[int]
    text: str  # line content without +/- prefix
```

Listing C.8: Diff line data structure

```
_DEF_FUNC_RE      = re.compile(r"^\s*def\s+([A-Za-z_]\w*)\s*\(")
_DEF_CLASS_RE     = re.compile(r"^\s*class\s+([A-Za-z_]\w*)\s*[:\(]")
_ASSIGN_CONST_RE  = re.compile(r"^\s*([A-Z_][A-Z0-9_]*)\s*=")
_ASSIGN_SELF_RE   = re.compile(r"^\s*self\.([A-Za-z_]\w*)\s*=")
```

Listing C.9: Regular expressions for strong symbol extraction from diff lines

Only lines with `kind` equal to `"add"` or `"del"` are scanned. Each match is stored with auditable evidence (file path, line number, code snippet).

```python
def extract_diff_symbols(patch: str) -> List[Dict[str, Any]]:
    symbols = []
    for line in parse_unified_diff(patch):
        if line.kind == "ctx":
            continue
        for pattern in [_DEF_FUNC_RE, _DEF_CLASS_RE,
                        _ASSIGN_CONST_RE, _ASSIGN_SELF_RE]:
            m = pattern.match(line.text)
            if m:
                symbols.append({
                    "name": m.group(1),
                    "path": line.path,
                    "lineno": line.new_lineno or line.old_lineno,
                    "snippet": line.text.strip()[:120],
                })
    return symbols
```

Listing C.10: Strong symbol extraction from diff

C.3.2 Symbol Normalization

All symbol comparisons are performed over a normalized variant set that accounts for naming convention differences between issue descriptions, code, and LLM output. Because normalization increases recall at the cost of potential over-matching, the resulting variants are used only as grounding candidates and are later filtered through the full grounding-strength procedure.

```python
def normalize_symbol(symbol: str) -> Set[str]:
    variants = {symbol}

    # Split dotted names: Class.method -> {Class, method}
    if "." in symbol:
        parts = symbol.split(".")
        variants.update(parts)
        # Strip self/cls prefix
        if parts[0] in ("self", "cls") and len(parts) > 1:
            variants.add(parts[1])

    # CamelCase -> snake_case and vice versa
    expanded = set()
    for v in list(variants):
        # CamelCase to snake_case
        snake = re.sub(r"(?<!^)(?=[A-Z])", "_", v).lower()
        expanded.add(snake)
        # snake_case to CamelCase
        camel = "".join(w.capitalize() for w in v.split("_"))
        expanded.add(camel)
    variants |= expanded

    return variants
```

Listing C.11: Symbol normalization variant generation

C.3.3 Grounding Strength Calculation

The grounding function evaluates each target symbol against the three levels in order, returning the strongest match found. Higher numeric levels correspond to stronger grounding.

```
@dataclass
class GroundingResult:
    status: str    # "strong" | "weak_file" | "weak_ref" | "none"
    level: int     # 3=strong, 2=weak_file, 1=weak_ref, 0=none
    matched_symbols: Set[str]
    unmatched_symbols: Set[str]
    evidence: List[Dict[str, Any]]

def calculate_grounding_strength(
    claim: Dict[str, Any],
    patch: str,
    *,
    touched_files_text: Optional[Dict[str, str]] = None,
    code_context: Optional[str] = None,
) -> GroundingResult:
    target_syms = claim.get("target_symbols", [])
    diff_syms = extract_diff_symbols(patch)
    matched, evidence = set(), []

    # Level 3: Strong grounding (diff symbols)
    diff_variants = {}
    for ds in diff_syms:
        diff_variants[ds["name"]] = normalize_symbol(ds["name"])
    for ts in target_syms:
        ts_variants = normalize_symbol(ts)
        for ds_name, ds_vars in diff_variants.items():
            if ts_variants & ds_vars:
                matched.add(ts)
                evidence.append({"symbol": ts, "matched_to": ds_name,
                                 "level": "strong"})
                break

    if matched:
        return GroundingResult("strong", 3, matched,
                               set(target_syms) - matched, evidence)

    # Level 2: Weak file grounding
    if touched_files_text:
        full_text = " ".join(touched_files_text.values()).lower()
        for ts in target_syms:
            for v in normalize_symbol(ts):
                if v.lower() in full_text:
```

```
                    matched.add(ts); break
        if matched:
            return GroundingResult("weak_file", 2, matched,
                                   set(target_syms) - matched, evidence
                                       )

    # Level 1: Weak reference grounding
    if code_context:
        ctx_lower = code_context.lower()
        for ts in target_syms:
            for v in normalize_symbol(ts):
                if v.lower() in ctx_lower:
                    matched.add(ts); break
        if matched:
            return GroundingResult("weak_ref", 1, matched,
                                   set(target_syms) - matched, evidence
                                       )

    return GroundingResult("none", 0, set(), set(target_syms), [])
```

Listing C.12: Grounding strength calculation

C.4 Agentic Loop Implementation

C.4.1 Guardrail Checks

The guardrail module evaluates four checks before test generation. Each check returns a structured result indicating whether it passed and whether a failure should be treated as blocking.

```
@dataclass
class GuardrailCheckResult:
    name: str       # "import" | "signature" | "fixture" | "probe"
    passed: bool
    blocking: bool  # whether failure prevents test generation
    details: Dict[str, Any]

@dataclass
class GuardrailResult:
    ok: bool                            # all blocking checks passed
    reason: Optional[str]               # first failing check label
    context: Dict[str, Any]             # module info, signatures
    checks: List[GuardrailCheckResult]
```

Listing C.13: Guardrail result data structures

The **signature check** is the most informative guardrail. When the target module can be imported, it introspects each target symbol to extract its call signature, required positional arguments, and whether it is an instance method.

```python
def _check_signature(module, symbol_name: str) -> Dict[str, Any]:
    obj = getattr(module, symbol_name, None)
    if obj is None:
        return {"found": False}
    try:
        sig = inspect.signature(obj)
    except (ValueError, TypeError):
        return {"found": True, "signature": None}

    params = list(sig.parameters.values())
    required = [
        p.name for p in params
        if p.default is inspect.Parameter.empty
        and p.kind in (p.POSITIONAL_ONLY, p.POSITIONAL_OR_KEYWORD)
    ]
    is_method = bool(required) and required[0] in ("self", "cls")
    if is_method:
        required = required[1:]  # exclude self/cls

    return {
        "found": True,
        "signature": str(sig),
        "required_args": required,
        "is_method": is_method,
    }
```

Listing C.14: Signature introspection during guardrail check

The import and probe checks are non-blocking by design, since many project-specific modules require dependencies only available inside the execution container. If a signature check failed in a previous iteration, it is downgraded to non-blocking on retry to allow the LLM to attempt alternative import paths.

C.4.2 Dynamic Planning Playbooks

The dynamic planner selects a playbook based on the diagnosis label from the previous failed iteration. Each playbook provides corrective preconditions, fallback strategies, and observable hints that are appended to the existing plan.

C.4.3 Anti-Pattern Detection

Before feeding error information back to the LLM, the pattern detector scans the generated test code for common anti-patterns. Four detectors are applied.

```python
def detect_common_antipatterns(
    code: str, error: str = ""
) -> List[Dict[str, Any]]:
    patterns = []

```

Diagnosis Label	Playbook Action
signature_check	Use package-level imports; avoid deep submodule paths.
import_check_failed	Use import patterns from issue code examples.
import_error	Try relative imports or fully qualified package names.
internal_import_error	Use only public APIs; no _pytest.* imports.
signature_mismatch	Check required positional arguments from grounding evidence.
fixture_missing	Use only standard pytest fixtures (tmp_path, monkeypatch, capsys).
assertion_failure	Assert directly on the then clause; relax implementation-specific checks.
non_discriminative	Ensure the observable differentiates buggy vs. patched behavior.
environment_error	Container issue; no changes to test code.

Table C.4.3: Dynamic planner playbooks keyed by diagnosis label.

```
    # 1. Fixture misuse: pytest.tmpdir_factory, pytest.monkeypatch, etc.
    for line in code.splitlines():
        for fixture in ("tmpdir", "tmpdir_factory", "monkeypatch",
                        "capsys", "capfd", "tmp_path"):
            if f"pytest.{fixture}" in line:
                patterns.append({"type": "fixture_attribute_error",
                                 "fixture_name": fixture, "line": line})

    # 2. Internal imports: from _pytest.* import ...
    for m in re.finditer(
        r"from\s+((?:\w+\.)*_\w+(?:\.\w+)*)\s+import\s+(\w+)", code
    ):
        patterns.append({"type": "internal_import",
                         "module": m.group(1),
                         "imported_names": m.group(2)})

    # 3. Non-existent pytest API: pytest.CapitalizedName
    for m in re.finditer(r"pytest\.([A-Z]\w+)", code):
        patterns.append({"type": "nonexistent_pytest_api",
                         "api_name": m.group(1)})

    # 4. Incorrect fixture creation: MonkeyPatch(), TempdirFactory()
    for m in re.finditer(
        r"(TempdirFactory|MonkeyPatch|CaptureFixture)\s*\(", code
    ):
        patterns.append({"type": "incorrect_fixture_creation",
                         "class_name": m.group(1)})
```

```
    return patterns
```

Listing C.15: Anti-pattern detection

C.4.4 Failure Diagnosis

The diagnostics module and the verification classifier (Section C.4.5) serve different purposes. The diagnostics module assigns actionable failure categories that guide the dynamic planner's refinement, whereas the verification classifier maps BUG/GOLD execution outcomes to the semantic labels reported by the layer (e.g., `VALID`, `NON_DISCRIMINATIVE`). The diagnostics classification inspects the execution results, the test source code, and any detected anti-patterns to produce a structured diagnosis.

```python
@dataclass
class Diagnosis:
    label: str                # e.g. "import_error", "overconstrained"
    details: str              # human-readable explanation + remediation
    patterns: List[Dict]      # detected anti-patterns

def classify_verification(
    result: Dict, generated_code: str = ""
) -> Diagnosis:
    bug_status  = result.get("bug", {}).get("status", "ERROR")
    gold_status = result.get("gold", {}).get("status", "ERROR")
    stderr = (result.get("gold", {}).get("stderr", "")
              + result.get("bug", {}).get("stderr", ""))

    patterns = detect_common_antipatterns(generated_code, stderr)

    # Success
    if bug_status == "FAIL" and gold_status == "PASS":
        return Diagnosis("success", "Test discriminates.", patterns)

    # Non-discriminative (early exit)
    if bug_status == "PASS" and gold_status == "PASS":
        return Diagnosis("non_discriminative",
            "Both versions pass. Claim may not capture the bug.",
            patterns)

    # Environment error (broken pytest installation)
    if "_pytest/_version" in stderr or "pytest/__init__.py" in stderr:
        return Diagnosis("environment_error",
            "Container pytest is broken; not a test code issue.",
            patterns)

    # Internal import
    if re.search(r"from _pytest|import _pytest", generated_code):
        return Diagnosis("internal_import_error",
            "Test imports private pytest internals.", patterns)
```

```

    # Import error
    if "ImportError" in stderr or "ModuleNotFoundError" in stderr:
        return Diagnosis("import_error",
            "Cannot import target module or symbol.", patterns)

    # Signature mismatch
    if "missing 1 required positional argument" in stderr:
        return Diagnosis("signature_mismatch",
            "Function called with wrong arguments.", patterns)

    # Overconstrained
    if bug_status == "FAIL" and gold_status == "FAIL":
        return Diagnosis("overconstrained",
            "Test fails on both versions; assertions too strict.",
            patterns)

    # Assertion failure (gold fails assertion)
    if gold_status == "FAIL" and "AssertionError" in stderr:
        return Diagnosis("assertion_failure",
            "Gold version fails an assertion; expected values wrong.",
            patterns)

    # Fixture missing
    if "fixture" in stderr.lower() and "not found" in stderr.lower():
        return Diagnosis("fixture_missing",
            "Test references unavailable fixtures.", patterns)

    # Default: unresolved
    return Diagnosis("unresolved",
        "Execution error or timeout; inconclusive.", patterns)
```

Listing C.16: Failure diagnosis data structure and classification

C.4.5 Verification Classification

The dual-environment execution results are classified by comparing the BUG and GOLD statuses. Listing C.17 shows the classification function that produces the labels described in Table 5.5.5.

```
def classify(bug_status: str, gold_status: str) -> Tuple[str, str]:
    if bug_status == "FAIL" and gold_status == "PASS":
        return "VALID", "Test discriminates correctly."
    if bug_status == "PASS" and gold_status == "PASS":
        return "NON_DISCRIMINATIVE", "Test passes on both versions."
    if bug_status == "FAIL" and gold_status == "FAIL":
        return "OVERCONSTRAINED", "Test fails on both versions."
    if bug_status == "PASS" and gold_status == "FAIL":
        return "INVERTED", "Test logic is inverted."
    return "UNRESOLVED", "Execution error or timeout."
```

Listing C.17: Verification classification

C.4.6 Loop Termination Logic

In each retry iteration, the loop returns to the planning stage rather than directly regenerating the test code. This reflects the use of both a static planner, which constructs the initial execution plan, and a dynamic planner, which refines that plan using the diagnosis produced by the previous iteration.

The agentic loop runs for at most K iterations (default $K = 3$). The stuck-loop detector is applied only to repeated guardrail failures: if the same blocking guardrail failure occurs three consecutive times, the loop terminates early. Repeated post-execution diagnosis labels are recorded but do not by themselves trigger early termination.

```
class AgenticClosedLoop:
    def run(self, max_attempts: int = 3) -> Dict[str, Any]:
        diagnosis = None
        consecutive_same = 0
        last_label = None

        for attempt in range(max_attempts):
            # Plan (static + dynamic refinement)
            plan = self.static_planner.build(self.context)
            if diagnosis is not None:
                plan = self.dynamic_planner.refine(plan, diagnosis)

            # Guardrails
            guard = self.guardrails.evaluate(plan)
            if not guard.ok:
                diagnosis = Diagnosis(guard.reason, guard.context, [])
                # Stuck-loop detection (guardrail failures only)
                if guard.reason == last_label:
                    consecutive_same += 1
                    if consecutive_same >= 3:
                        return {"success": False,
                                "failure_reason": "stuck_in_loop"}
                else:
                    consecutive_same = 1
                last_label = guard.reason
                continue

            # Generate sketch -> code -> verify
            sketch = self.sketcher.generate(plan, diagnosis)
            code = self.writer.generate(plan, sketch, diagnosis)
            result = self.verifier.run(code)

            # Classify
            diagnosis = classify_verification(result, code)

            if diagnosis.label == "success":
                return {"success": True, "attempts": attempt + 1}
            if diagnosis.label == "non_discriminative":
                return {"success": False,
```

```
                        "failure_reason": "non_discriminative"}

            last_label = diagnosis.label

        return {"success": False, "failure_reason": "max_attempts"}
```

Listing C.18: Agentic loop main control flow

C.5 Configuration Defaults

These defaults were selected to balance semantic coverage, execution cost, and reproducibility across repositories in the experimental subset. Table C.5.4 summarizes the key configuration constants used throughout the semantic verification layer.

Parameter	Default	Description
`eligibility.threshold`	2	Minimum score for issue eligibility.
`context.max_chars`	22 000	Maximum code context length (characters).
`llm.temperature`	0.0	Sampling temperature for claim extraction.
`llm.max_tokens`	2 048	Maximum output tokens for claim extraction.
`require_grounding`	True	Discard claims with grounding status `none`.
`min_claim_score`	2	Minimum composite score for claim inclusion.
`max_claims_per_issue`	6	Maximum claims forwarded to test generation.
`evidence_score_cap`	3	Maximum evidence verification score.
`max_attempts`	3	Maximum iterations of the agentic loop.
`timeout_s`	300	Per-test execution timeout (seconds).
`stuck_threshold`	3	Consecutive identical guardrail failures before exit.

Table C.5.4: Default configuration parameters for the semantic verification layer.

Taken together, these components implement the semantic verification layer as a grounded, iterative process that translates natural-language issue descriptions into executable behavioral tests. This appendix complements the architectural description in Section 5.5 by documenting the concrete heuristics and control-flow decisions used in the prototype implementation.

C.5.1 Prompt Template Example (Claim Extraction)

The following code listing C.19 shows the prompt structure used for the claim-extraction task.

```

You are a software verification expert analyzing a GitHub issue.

Your task is to extract testable behavioral claims describing what the
    code
```

```
SHOULD do after the bug is fixed.

=====================
ISSUE DESCRIPTION
=====================
{{ problem_statement }}

=====================
RELEVANT SOURCE CODE
(current buggy context)
=====================
{{ code_context }}

=====================
WHAT IS A VALID CLAIM
=====================
- Concrete Given / When / Then
- Observable outcome:
  - return value
  - raised or NOT raised exception
  - state change
- target_symbols must exist in the code shown above
- Prefer the most specific symbol actually modified in the diff (helper functions,
  nested functions, private methods, etc.), even if the issue title references a
  higher-level API. Include multiple target symbols if several helpers participate.
- evidence.spans must be EXACT quotes from the issue text
- Claims must be testable with a unit test

INVALID CLAIMS (DO NOT GENERATE)
- Vague: "should work", "should handle properly"
- No observable outcome
- Hallucinated symbols not present in code
- Only referencing a top-level public API when the change clearly happens inside a helper
- Invented evidence not present in the issue

=====================
CLAIM TYPES
=====================
- return
- exception
- invariant
- state_change

IMPORTANT NORMALIZATION RULES
- Do NOT narrow or over-specify the expected behavior beyond what the issue states.
  If the issue provides multiple acceptable forms, the claim MUST
```

```
     preserve that flexibility.

======================
OUTPUT FORMAT
======================
Return ONLY a JSON array.
Do NOT include markdown.
Do NOT include explanations.
If the issue is too vague, return [].

Each item must follow this schema exactly:

[
  {
    "claim_id": "C1",
    "claim_type": "return | exception | invariant | state_change",
    "claim_text": "One sentence expected behavior",
    "given": "Specific precondition",
    "when": "Specific action with real function name",
    "then": "Specific observable outcome",
    "target_symbols": ["real_function_or_class_name"],
    "confidence": "high | medium | low",
    "evidence": {
      "spans": ["EXACT quote from the issue text"]
    }
  }
]

BEGIN JSON ARRAY:
```

Listing C.19: Claim Extraction Prompt Example

Appendix D

Project Alignment with SDG

This project is closely related to several of the Sustainable Development Goals (SDGs)[43] proposed by world leaders for the year 2030, despite its highly technical and research-oriented nature.

The goal that shows the strongest alignment with this project is Goal 9: **"Industry, Innovation and Infrastructure"**. The following targets represent the points of greatest overlap with the developed work:

- 9.1 Develop quality, reliable, sustainable and resilient infrastructure, including regional and transborder infrastructure, to support economic development and human well-being

- 9.5 Enhance scientific research and upgrade the technological capabilities of industrial sectors, encouraging innovation and increasing research and development capacity

During the development of this project, research has been conducted on advanced techniques in artificial intelligence and software engineering, with a particular focus on improving the reliability of large language model (LLM) generated code. The proposed multi-layer verification harness introduces a structured framework that integrates static analysis, dynamic execution, and semantic validation, contributing to the development of more robust and trustworthy digital infrastructures.

In addition, this work contributes to the advancement of research in AI-driven software engineering by proposing a novel approach to evaluating automatically generated code patches. By addressing limitations of traditional test-suite-based validation and introducing claim-based semantic verification, the project supports innovation in the field and enhances the understanding of how LLMs can be reliably integrated into real-world development workflows.

Bibliography

[1] Juyong Jiang et al. “A Survey on Large Language Models for Code Generation”. In: *ACM Transactions on Software Engineering and Methodology* 35.2 (Jan. 2026), pp. 1–72. ISSN: 1557-7392. DOI: 10.1145/3747588. URL: http://dx.doi.org/10.1145/3747588.

[2] Carlos E. Jimenez et al. “SWE-bench: Can Language Models Resolve Real-World GitHub Issues?” In: *The Twelfth International Conference on Learning Representations (ICLR)*. 2024. URL: https://openreview.net/forum?id=VTF8yNQM66.

[3] You Wang, Michael Pradel, and Zhongxin Liu. “Are “Solved Issues” in SWE-bench Really Solved Correctly? An Empirical Study”. In: *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*. 2025. URL: https://arxiv.org/abs/2503.15223.

[4] Kunal Taneja and Tao Xie. “DiffGen: Automated Regression Unit-Test Generation”. In: *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*. 2008, pp. 407–410. DOI: 10.1109/ASE.2008.60.

[5] Qi Xin and Steven P. Reiss. “Identifying Test-Suite-Overfitted Patches through Test Case Generation”. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2017, pp. 226–236. DOI: 10.1145/3092703.3092718.

[6] Sungmin Kang, Juyeon Yoon, and Shin Yoo. “Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction”. In: 2023. arXiv: 2209.11515 [cs.SE]. URL: https://arxiv.org/abs/2209.11515.

[7] Barry Boehm and Victor Basili. “Software Defect Reduction Top 10 List”. In: *IEEE Computer* 34.1 (2001), pp. 135–137. DOI: 10.1109/2.962984.

[8] Boyang Yang et al. *A Survey of LLM-based Automated Program Repair: Taxonomies, Design Paradigms, and Applications*. 2025. arXiv: 2506.23749 [cs.SE]. URL: https://arxiv.org/abs/2506.23749.

[9] Mark Chen et al. “Evaluating Large Language Models Trained on Code”. In: (2021). arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374.

[10] OpenAI Preparedness Team. *Introducing SWE-bench Verified*. OpenAI Technical Report. 2024. URL: https://openai.com/index/introducing-swe-bench-verified/.

[11] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. "RepairAgent: An Autonomous, LLM-Based Agent for Program Repair". In: 2024. arXiv: 2403.17134 [cs.SE]. URL: https://arxiv.org/abs/2403.17134.

[12] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62.

[13] Gordon Fraser and Andrea Arcuri. "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. 2011, pp. 416–419. DOI: 10.1145/2025113.2025179.

[14] Yinghao Chen et al. "ChatUniTest: A Framework for LLM-Based Test Generation". In: 2024. arXiv: 2305.04764 [cs.SE]. URL: https://arxiv.org/abs/2305.04764.

[15] Yifan Zhang et al. "CoverAgent: Coverage-Guided Test Generation Using Large Language Models". In: *arXiv preprint arXiv:2403.16218* (2024). URL: https://arxiv.org/pdf/2403.16218.

[16] Bei Chen, Fengji Zhang, et al. "CodeT: Code Generation with Generated Tests". In: *The Eleventh International Conference on Learning Representations (ICLR)*. 2023. URL: https://openreview.net/forum?id=ktrw68Cmu9c.

[17] Ansong Ni et al. "LEVER: Learning to Verify Language-to-Code Generation with Execution". In: *Proceedings of the 40th International Conference on Machine Learning (ICML)*. 2023.

[18] Michael D. Ernst et al. "The Daikon System for Dynamic Detection of Likely Invariants". In: *Science of Computer Programming* 69.1-3 (2007), pp. 35–45. DOI: 10.1016/j.scico.2007.01.015.

[19] John Yang et al. "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering". In: 37 (2024). Ed. by A. Globerson et al., pp. 50528–50652. DOI: 10.52202/079017-1601. URL: https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

[20] Chunqiu Steven Xia et al. "Agentless: Demystifying LLM-based Software Engineering Agents". In: (2024). arXiv: 2407.01489 [cs.SE]. URL: https://arxiv.org/abs/2407.01489.

[21] Xinyun Chen et al. "Teaching Large Language Models to Self-Debug". In: (2023). arXiv: 2304.05128 [cs.CL]. URL: https://arxiv.org/abs/2304.05128.

[22] He Ye, Matias Martinez, and Martin Monperrus. "Automated Patch Assessment for Program Repair at Scale". In: *Empirical Software Engineering* 26.20 (2021). DOI: 10.1007/s10664-020-09920-w.

[23] Manish Bhatt et al. "CyberSecEval 2: A Wide-Ranging Cybersecurity Evaluation Suite for Large Language Models". In: 2024. arXiv: 2404.13161 [cs.CR]. URL: https://arxiv.org/abs/2404.13161.

[24] Anisha Agarwal et al. “Copilot Evaluation Harness: Evaluating LLM-Guided Software Programming”. In: (2024). arXiv: 2402.14261 [cs.SE]. URL: https://arxiv.org/abs/2402.14261.

[25] Henry Gordon Rice. “Classes of Recursively Enumerable Sets and Their Decision Problems”. In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366. DOI: 10.1090/S0002-9947-1953-0053041-6.

[26] Zhangyin Feng et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”. In: 2020. arXiv: 2002.08155 [cs.CL]. URL: https://arxiv.org/abs/2002.08155.

[27] Yue Wang et al. “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation”. In: 2021. arXiv: 2109.00859 [cs.CL]. URL: https://arxiv.org/abs/2109.00859.

[28] Django Software Foundation. *Django Web Framework*. https://www.djangoproject.com. 2024.

[29] John D. Hunter. “Matplotlib: A 2D Graphics Environment”. In: *Computing in Science and Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[30] Fabian Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (Jan. 2012).

[31] Woosuk Kwon, Sukmin Kim, et al. “Efficient Memory Management for Large Language Model Serving with vLLM”. In: *arXiv preprint arXiv:2309.06180* (2023).

[32] PyCQA. *Pylint Documentation*. https://pylint.org.

[33] PyCQA. *Flake8 Documentation*. https://flake8.pycqa.org.

[34] Michele Lacchia. *Radon: Code Metrics in Python*. https://radon.readthedocs.io.

[35] Jukka Lehtosalo et al. *Mypy: Optional Static Typing for Python*. https://mypy-lang.org.

[36] OpenStack Security Project. *Bandit: Security oriented static analyzer for Python*. https://bandit.readthedocs.io. 2024.

[37] Pytest development team. *Pytest Documentation*. https://docs.pytest.org/. 2024.

[38] Andy B. Yoo, Morris A. Jette, and Mark Grondona. “SLURM: Simple Linux Utility for Resource Management”. In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4.

[39] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. “Singularity: Scientific Containers for Mobility of Compute”. In: *PLOS ONE* 12.5 (2017), e0177459. DOI: 10.1371/journal.pone.0177459.

[40] Streamlit Inc. *Streamlit: The fastest way to build data apps*. https://streamlit.io. 2024.

[41] Plotly Technologies Inc. *Plotly: Collaborative Data Science*. https://plotly.com. 2024.

[42] Jake VanderPlas et al. "Altair: Interactive Statistical Visualizations for Python". In: *Journal of Open Source Software* 3.32 (2018), p. 1057. DOI: 10.21105/joss.01057.

[43] Organización de las Naciones Unidas (ONU). *Objetivos de Desarrollo Sostenible*. URL: https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/.