



Needle in the Repo: Diagnosing Maintainability Failures in AI-Generated Repository Edits

Haichao Zhu^{*†}
Qualcomm
Santa Clara, USA
haiczhu@qti.qualcomm.com

Qian Zhang^{*✉}
University of California, Riverside
Riverside, USA
qzhang@cs.ucr.edu

Jiyuan Wang
Tulane University
New Orleans, USA
wjiyuan@tulane.edu

Zhaorui Yang
University of California Riverside
Riverside, USA
zyang247@ucr.edu

Yuxin Qiu
University of California Riverside
Riverside, USA
yuxin.qiu@email.ucr.edu

Abstract

AI coding agents can now complete complex programming tasks, but existing evaluations largely emphasize behavioral correctness and often overlook maintainability risks such as weak modularity or testability. Automation without accountability shifts invisible cost downstream because hidden maintainability failures incurred today are deferred to the developers who must extend, debug, and sustain these systems tomorrow, raising significant technical and organizational concerns for software engineering.

We present Needle in the Repo (NITR), a diagnostic, probe-and-oracle framework for evaluating whether behaviorally correct repository edits preserve maintainable structure. The key idea is to distill recurring software engineering wisdom into *controlled probes* embedded in small, realistic multi-file codebases, with each probe designed so that success depends primarily on one targeted maintainability dimension. Each probe is paired with a hidden evaluation harness that combines (i) functional tests for required behavior, and (ii) structural oracles that encode the targeted maintainability constraint and return interpretable diagnoses.

Using NITR, we *extensively* evaluate 23 coding configurations across GPT, Claude, Gemini, and Qwen families in both direct-inference and agent-based settings. We find that current AI coding systems remain far from robust. On average, all AI coding configurations solve only 36.2% of cases, the best reaches 57.1%, and performance drops from 53.5% on micro cases to 20.6% on multi-step cases. The hardest pressures are architectural rather than local edits, especially dependency control (4.3%) and responsibility decomposition (15.2%). Moreover, 64/483 outcomes (13.3%) pass all functional tests yet fail the structural oracle. Under our harness, agent-mode configurations improve average performance from 28.2% to 45.0%, but do not eliminate the architectural failures.

These results show that progress in code generation is not yet progress in maintainable code evolution, and that NITR exposes a critical failure surface largely missed by conventional evaluation.

CCS Concepts

• Software and its engineering → Maintaining software; Software evolution; • Social and professional topics → Codes of ethics.

Keywords

Software Maintenance, Software Evolution, Software Engineering Benchmark

ACM Reference Format:

Haichao Zhu, Qian Zhang, Jiyuan Wang, Zhaorui Yang, and Yuxin Qiu. 2026. Needle in the Repo: Diagnosing Maintainability Failures in AI-Generated Repository Edits. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*, October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3832783.3834401>

1 Introduction

Artificial intelligence (AI) is rapidly reshaping software development, demonstrating strong capabilities in code understanding, generation, execution, and iterative debugging [21, 37, 39]. We use the term *coding agents* to refer to large language model (LLM)-based systems equipped with scaffolding such as repository navigation [31, 39], code editing [37, 39], program execution [36], and feedback-driven repair [8, 34]. These agents now operate directly inside real development environments, making their evaluation increasingly important for understanding what kinds of software engineering (SE) work they can actually support.

The Gap: Completion is Not Construction. Recent benchmarks have made substantial progress by moving beyond single-file completion toward repository-level tasks [21, 25, 27]. For example, SWE-bench and its variants [21, 40] evaluate LLMs on issue resolution across 12 GitHub repositories. However, these evaluations still define success primarily in terms of *behavioral correctness*, operationalized as pass/fail performance on predefined test cases. This leaves a fundamental gap between what current benchmarks measure and what SE practice actually requires, illustrated in Figure 1.

^{*}Equal co-first authors.

[†]This research was conducted before the author joined Qualcomm.



This work is licensed under a Creative Commons Attribution 4.0 International License. ASE '26, Munich, Germany

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
<https://doi.org/10.1145/3832783.3834401>

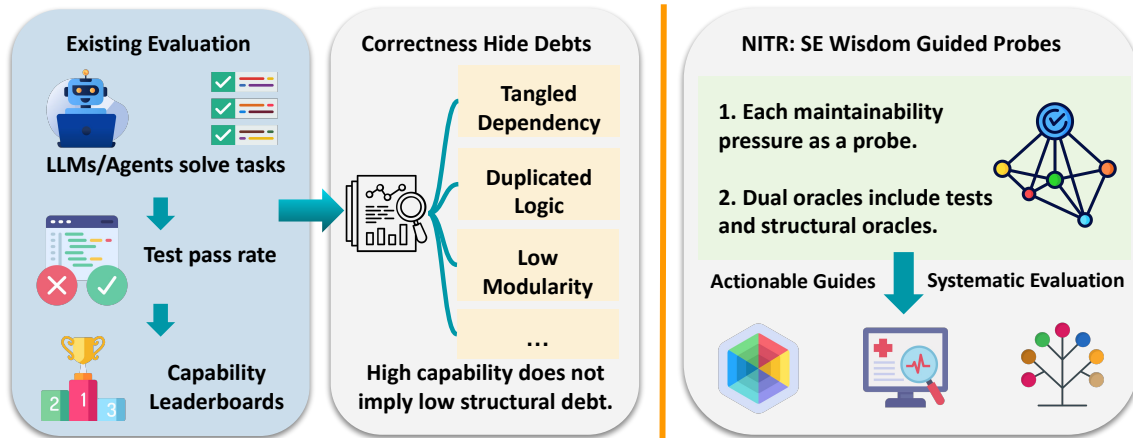


Figure 1: Existing evaluations rank models by behavioral success, but leaderboard performance says little about maintainability risk. NITR uses curated probes and structural oracles to expose failure modes beyond test passing.

First, test pass/fail is a narrow view of software quality. A patch may satisfy the target tests while still duplicating logic, bypassing an existing abstraction, or spreading a small feature across unrelated modules, thereby making the repository harder to extend and maintain over time. Recent studies of non-functional quality in LLM-generated code likewise find that behavioral success does not guarantee maintainability, reliability, or broader quality attributes [29, 35]. This concern also appears in practitioner discussion. For example, one recent Reddit comment [4] observed that “The real failure mode I keep seeing is maintenance.”

Second, test-centric evaluations provide weak diagnostic insight. They indicate whether agents completed a task, but not which maintainability capabilities they handled well or failed to preserve. Previous work has shown that correctness-based benchmarks often miss specific failure modes and overlook code-quality concerns that matter in practice [9, 19, 45]. As a result, they rank models by outcome, but reveal little about where current LLMs still fall short.

This Work. This work addresses a core limitation of current code-agent evaluation: passing repository tasks does not tell us whether the resulting code remains maintainable, or which maintainability capabilities current LLMs fail to preserve. We present *NEEDLE IN THE REPO* (NITR), a probe-and-oracle framework for evaluating maintainability preservation in AI-generated repository edits. Throughout this paper, we use *maintainability failure* to refer to code that is behaviorally correct for the current task but makes future evolution harder to extend, test, or reason about safely.

Our central insight is that software engineering wisdom can be distilled into small, repository-grounded probes with explicit design boundaries and executable structural checks. NITR operationalizes 9 maintainability dimensions through targeted C++ repository probes: Change Locality, Reuse and Repo Awareness, Responsibility Decomposition, Extension Structure, Interface and Substitutability Discipline, Dependency Control, Testability and Determinism, State Ownership and Lifecycle, and Side-Effect Isolation.

NITR comprises 21 curated C++ repository probes spanning nine maintainability dimensions. Each case encodes one primary repository-evolution pressure through a natural multi-file change

request and is paired with hidden functional tests and structural probes, enabling diagnosis beyond test passing.

Key Findings. Using NITR, we evaluate 23 coding configurations spanning GPT, Claude, Gemini, and Qwen families in both direct-inference and agent-mode settings.

We present, to our knowledge, the first probe-and-oracle study focused on maintainability preservation in AI-generated repository edits and report three findings. First, current coding systems remain far from robust at maintainability-preserving repository evolution: the average configuration solves only 36.2% of cases, the best reaches 57.1%, and five probes are unsolved by all configurations. Second, current systems struggle most not with isolated edits, but with changes that must preserve deeper repository structure, including dependency control (4.3% pass rate), responsibility decomposition (15.2%), and abstraction-respecting extension (26.1%). Third, test passing is an unreliable proxy for maintainability: 64/483 outcomes (13.3%) are behaviorally correct yet structurally wrong, and although agent-mode configurations under our harness raises average performance from 28.2% to 45.0%, it does not eliminate these core failures. Together, these results show that NITR reveals hidden structural debt that conventional evaluations largely miss.

This work makes three contributions:

- NITR turns maintainability-preserving repository edit evaluation into a controlled object of study by pairing curated repository probes with hidden structural oracles.
- We provide the first probe-and-oracle empirical study of maintainability preservation in AI-generated repository edits, showing that test-passing often misses structural failure and that the hardest pressures are architectural rather than local.
- We release NITR as an open-source suite to support future research on maintainability-aware coding agents and repository-level software engineering evaluation.

NITR is not intended as a universal taxonomy or a naturalistic sample of all software tasks; it is a diagnostic suite designed to make specific maintainability pressures observable, executable, and comparable. In other words, it does not attempt to cover all

```

1 // Non-maintainable: submitted by 17/23 agents. Each new type
2 // requires duplicating a definition (change amplification).
3 namespace solid::case001 {
4     int add(int a, int b) { return a + b; }
5     float add(float a, float b) { return a + b; }
6     double add(double a, double b) { return a + b; }
7     long long add(long long a, long long b) { return a + b; }
8 }
9
10 // Maintainable: a single generic definition handles all types.
11 // Adding support for a new type requires zero code change.
12 namespace solid::case001 {
13     template <typename T>
14     T add(T a, T b) { return a + b; }
15 }

```

Listing 1: Two implementations of a multi-type add function. Both pass all unit tests, but the overloaded version (top) fails the structural probe because it introduces a new definition for each type added, amplifying change across the codebase. The template version (bottom) passes both unit tests and the structural probe.

maintainability phenomena, but to make a set of recurring, practically important repository-evolution pressures precise enough to evaluate, compare, and analyze.

2 Motivating Example

Coding agents are often evaluated as if passing tests were sufficient evidence of code quality. The motivating example in case 001 of NITR shows why this assumption is too weak. In this task, a small utility function add is widely used across a codebase. The agent is asked, over three incremental steps, to extend it to support additional numeric types: first `int`, then `float` and `double`, and finally `long long`. Throughout all steps, the call sites in `app/main.cc` remain fixed and must not be modified.

Listing 1 shows two representative implementations submitted by agents. The *overloaded* implementation (lines 2–8) is the approach taken by 17 of 23 evaluated configurations. It introduces a separate, copy-pasted definition for every numeric type. Functionally, both implementations are equivalent: all unit tests pass. The tests check correct addition output for each required type; they do not inspect the structure of the implementation.

The structural probe in NITR, however, evaluates a different criterion. It checks that the implementation provides a *centralized reusable core* (detected via generic-function patterns such as a template) and that no more than one explicit per-type overload definition is present. The overloaded implementation (lines 2–8) violates both conditions and is therefore marked as failing, even though every functional test passes.

This failure has direct maintenance consequences. If the codebase later needs to support `unsigned int` or `float128`, the overloaded design requires adding another specialized definition and modifying the library interface. In a multi-step tasks, the problem becomes worse. Once an agent has introduced the overload pattern, later steps may continue extending that pattern rather than repairing the abstraction, compounding the original design mistake.

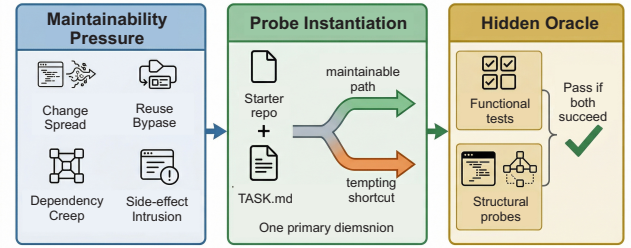


Figure 2: From maintainability pressure to diagnostic probe. NITR summarizes SE practices into 9 recurring repository-evolution pressures, instantiates each target pressure as a compact probe with starter code and an agent-facing task, and pairs the probe with functional and structural oracles.

This example illustrates the exact failure mode that NITR is designed to expose: code can be behaviorally correct yet structurally poor to evolve.

3 Maintainability Probe Design

We construct NITR in three parts as shown in Figure 2. First, we define a maintainability design space grounded in recurring software-evolution pressures (Section 3.1). Second, we instantiate each target pressure as a compact repository probe with starter code and an agent-facing task (Section 3.2). Third, we package each probe with hidden functional tests and structural oracles that distinguish maintainable solutions from tempting shortcut solutions (Section 3.3).

NITR is organized as a collection of expert-curated maintainability probes rather than generic coding tasks mined from GitHub that may already be familiar to LLMs. Each probe is a small repository-level scenario designed to expose one primary maintainability pressure under realistic software evolution. The evaluation target is whether AI’s chosen implementation strategy preserves maintainable structure. The resulting suite contains 21 C++ probes spanning 9 software engineering dimensions, including 10 micro probes and 11 multi-step probes detailed below.

3.1 From Principles to Maintainability Dimensions

Our starting point is that software engineering wisdom already provides a practical basis for judging whether code will remain easy to extend, modify, test, and integrate over time. Accordingly, NITR begins from the design principles and engineering practices that developers use to assess code quality in real repositories.

To define the construction design space of NITR, we reviewed established software engineering principles, including SOLID [28], together with maintainability practices such as code reuse [18, 22] and testability [17, 38].

From this author-driven consolidation, we derived the nine dimensions by consolidating recurrent repository-evolution pressures rather than by starting from a fixed taxonomy alone. Concretely, we first enumerated common structural failure pressures that arise during repository change, such as change amplification, duplicate-path introduction, responsibility leakage, dependency creep, test-seam erosion, lifecycle scattering, and side-effect intrusion. We

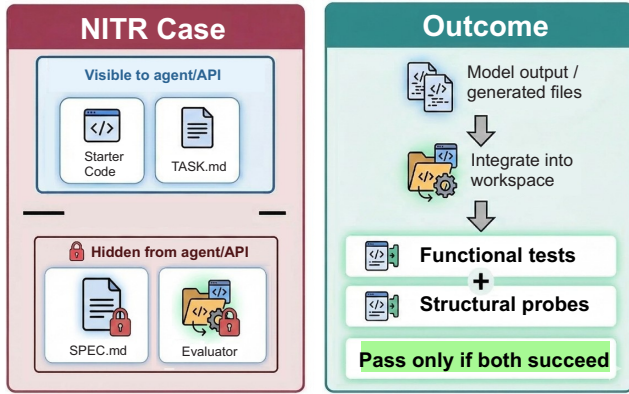


Figure 3: Anatomy of a maintainability probe. Each probe contains starter code, an agent-visible task (TASK.md), an author-facing specification (SPEC.md), and a hidden evaluator. During evaluation, the model sees only the starter code and TASK.md; the probe passes only if the generated edits satisfy both the functional tests and the structural probes.

then grouped these pressures into broader, reusable construction axes, using classical software-design principles such as SOLID as an organizing lens where appropriate. The resulting nine dimensions are therefore best understood as probe-construction axes for maintainability-relevant repository evolution, not as an exhaustive ontology of maintainability.

The nine dimensions include: (D1) *Change Locality*, whether a requested change remains localized rather than forcing scattered edits; (D2) *Reuse and Repo Awareness*, whether the solution reuses and adapts existing repository logic instead of re-implementing it in parallel; (D3) *Responsibility Decomposition*, whether distinct concerns remain separated rather than collapsing into one component; (D4) *Extension Structure*, whether new behavior is added through clean extension rather than ad hoc special cases; (D5) *Interface and Substitutability Discipline*, whether interfaces remain narrow, coherent, and safe for interchangeable implementations; (D6) *Dependency Control*, whether modules avoid unnecessary coupling and depend only on what they need; (D7) *Testability and Determinism*, whether the design preserves clean test seams and avoids hidden nondeterminism; (D8) *State Ownership and Lifecycle*, whether mutable state has clear ownership and lifetime boundaries and (D9) *Side-Effect Isolation*, whether effects such as I/O, logging, and global mutation are kept separate from core logic. Table 1 reports coverage at the dimension level, while the design matrix records the concrete failure pressures each dimension was designed to capture.

Each case declares exactly one *primary_dimension*, defined as the main maintainability pressure the case is designed to isolate. Optional *secondary_dimensions* record supporting pressures when needed. Assigning one primary dimension keeps each probe interpretable and supports balanced coverage across the design space, while still allowing secondary pressures to appear as context.

3.2 Instantiating Maintainability Probes

Given a target dimension, we instantiate it as a small, self-contained C++ repository probe. A probe is a jointly designed artifact consisting of a repository scenario, starter code, an agent-visible task, and hidden evaluation logic, as illustrated in Figure 3. Each probe is constructed so that the target maintainability pressure arises naturally, a shortcut solution remains plausible, and the difference between the two can be diagnosed automatically. Please note that this curation is deliberate rather than incidental. In mined tasks, the intended design boundary is often implicit, underspecified, or entangled with unrelated repository history, making maintainability judgment difficult to operationalize. NITR instead uses compact, authored repository scenarios where the maintainable path and the tempting shortcut are both behaviorally plausible, but structurally distinguishable. This enables interpretable diagnosis of edit strategy rather than only outcome success.

Step 1: Repository Scenario Selection. We first select a compact engineering change that naturally exposes the target maintainability pressure. The scenario should be realistic and narrow in scope, while still admitting two plausible implementation paths: one that preserves repository structure and one that satisfies the required behavior through a shortcut.

For example, the curated Case 021 targets D2: *Reuse and Repo Awareness*. The requested change is to add support for an inline filter form such as `status=open,priority=3`. The repository already supports filtering through an existing structured pipeline, so the core question is whether the new inline entry point reuses that pipeline or introduces a second parsing-and-validation path. Listing 2 shows the associated structural oracle.

Step 2: Starter-Code Shaping. We then shape the starter repository so that the intended maintainable path already exists in the codebase. In Case 021, the starter code already contains a repository-native filter representation, an existing parse/validate pipeline, and a designated validation module. This creates two clear solution paths. A maintainable solution parses the inline string only far enough to construct the existing internal representation and then delegates to the current pipeline. A shortcut solution instead introduces inline-only logic for numeric parsing, field interpretation, or error classification. These shortcut patterns are encoded directly in the hidden oracle checks in Listing 2, including checks for `std::stoi`, `std::isdigit`, duplicated field literals, and duplicated error literals.

Step 3: Agent-Facing Task Design. For each probe, we write a concise TASK.md as a natural engineering request. It specifies the required behavior, but does not reveal the maintainability dimension, the intended design move, or the hidden oracle logic.

In Case 021, the visible task asks the agent to add support for the inline filter form while preserving existing behavior. It does not say “reuse the validation module,” does not mention D2, and does not warn against introducing a helper such as `ParseInlineFilter` with its own integer parsing and field checks.

Step 4: Micro versus Multi-Step Probes. We use a micro probe when the targeted maintainability pressure can be exposed by a single localized change, and a multi-step probe when it only becomes visible under continued evolution. In a multi-step probe, the

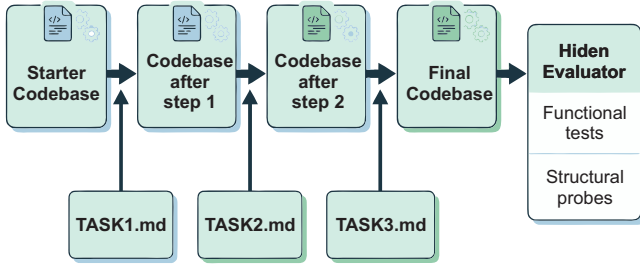


Figure 4: Multi-step probe execution. Each step is applied to the codebase produced by the previous step, so early design choices persist across later changes. The final codebase is then evaluated with tests and structural oracles.

model receives a sequence of task files, each applied to the code produced at the previous step, as illustrated in Figure 4. This makes early design decisions persistent across later changes.

Case 021 is a micro probe because the D2 failure is visible after one change. Once the agent adds the inline entry point, the solution either routes it through the existing pipeline or introduces inline-only parsing and validation logic. The hidden oracle can detect that shortcut immediately through patterns such as `std::stoi`, `std::isdigit`, duplicated field handling, or duplicated error classification, so no further evolution is needed to expose the difference. By contrast, dimensions such as *Dependency Control* or *Responsibility Decomposition* often require multi-step probes, because a design that looks acceptable at step 1 may only fail later when added features force one component to absorb logging, provider-specific logic, or extra dependencies.

3.3 Packaging Dual Oracles

Each probe is paired with two hidden evaluators: *functional tests*, which check the requested behavior, and *structural oracles*, which check whether the implementation preserves the intended maintainability constraint. A probe passes only if both succeed.

The structural oracle is probe-specific, but its design follows recurring patterns aligned with the nine dimensions. For *Change Locality* (D1), it checks whether a small requirement change remains localized. For *Reuse and Repo Awareness* (D2), it checks reuse of existing repository affordances and rejects duplicate implementation. For *Responsibility Decomposition* (D3), *Extension Structure* (D4), and *Dependency Control* (D6), it checks whether new behavior stays within the intended module boundaries rather than expanding central dispatch logic or introducing unnecessary coupling. For *Interface and Substitutability Discipline* (D5), it checks that interfaces remain narrow and that callers do not branch on concrete subtype identity. For *Testability and Determinism* (D7), it flags hidden wall-clock or randomness usage and verifies that the intended deterministic seam is preserved. For *State Ownership and Lifecycle* (D8), it checks that lifecycle-sensitive state transitions remain within the designated ownership boundary. For *Side-Effect Isolation* (D9), it ensures that logging, tracing, or explanation paths do not intrude into core decision logic. These checks are lightweight and targeted rather than full semantic program analyses, but they are

```

1 # Check 1: The inline entrypoint should not perform its own
2 # numeric parsing. In the intended design, integer validation
3 # stays in the existing filter-validation path rather than being
4 # reimplemented inline.
5 has_stoi = "std::stoi" in content
6 has_isdigit = "std::isdigit" in content
7
8 if has_stoi or has_isdigit:
9     fail("Suspicious duplicate numeric validation")
10
11 # Check 2: Repeating field and error literals outside the
12 # designated validation files often signals a shadow parser with
13 # its own lookup tables and error mapping logic.
14 has_inline_parser = "ParseInlineFilter" in content
15 field_hits = sum(literal in content for literal in FIELD_LITERALS)
16 error_hits = sum(literal in content for literal in ERROR_LITERALS)
17
18 if has_inline_parser and field_hits > 0 and error_hits > 1:
19     fail("Suspicious duplicated error classification")

```

Listing 2: Excerpt of the hidden structural oracle for Case 021. The individual string-level checks shown here are not used as standalone maintainability judgments; they function as case-specific signals within a jointly designed repository context.

sufficient to distinguish the intended maintainable solution family from common shortcut patterns.

Case 021 illustrates the oracle pattern. Functional tests alone cannot distinguish reuse of the existing filter pipeline from a shadow parser, so the structural oracle checks for duplicated parsing and validation outside the designated repository path. Listing 2 shows an excerpt: it flags uses of `std::stoi` or `std::isdigit`, rejects inline-only integer handling when `FilterValueKind::kInteger` appears together with `ParseInlineFilter`, and detects repeated field or error logic outside the existing validation flow. In this way, the oracle distinguishes reuse of repository affordances from parallel reimplementation.

Oracle validation. During case construction, each oracle was validated against intended maintainable solutions, representative shortcut solutions, and near-miss variants. We iterated until the oracle stably separated the intended solution family from targeted shortcuts under small implementation variation; representative intended, shortcut, and near-miss variants are included in the artifact.

3.4 Evaluation Protocol

For each probe, the harness creates a temporary workspace, applies the model-generated file edits, configures and builds the project with CMake, runs the hidden functional tests, and then executes the structural oracles. A probe is marked *pass* only if both the functional and structural evaluations succeed.

This strict conjunction is intentional. NITR is designed to expose a failure surface that ordinary test-centric evaluation misses: solutions that satisfy the requested behavior while introducing structural shortcuts. Table 1 summarizes the nine maintainability dimensions covered by NITR, and Table 2 provides the full probe inventory, including each case’s dimension, granularity, and aggregate outcome counts across the evaluated configurations.

4 Experimental Setup

We use NITR to study coding agents and answer questions:

Table 1: The nine maintainability dimensions used as probe-construction axes in NITR. Each case is assigned exactly one primary dimension. The last column reports how many cases in that dimension involve multiple steps.

Dim.	Name	#Cases	#Multi-step
D1	Change Locality	3	1
D2	Reuse & Repo Awareness	3	2
D3	Responsibility Decomposition	2	1
D4	Extension Structure	2	1
D5	Interface & Substitutability	2	1
D6	Dependency Control	2	1
D7	Testability & Determinism	3	2
D8	State Ownership & Lifecycle	2	1
D9	Side-Effect Isolation	2	1
Total		21	11

- **RQ1:** How well do contemporary coding systems perform on the NITR case suite?
- **RQ2:** Which maintainability dimensions are most challenging for current coding systems?
- **RQ3:** What structural failure patterns do current AI coding tools exhibit, and to what extent are these failures missed by functional tests alone?
- **RQ4:** Under our constrained evaluation harness, to what extent do agent-mode configurations outperform direct API-based configurations from the same model family on maintainability preservation?

Models and Agents. We evaluate 23 configurations from multiple providers: 11 agent-mode and 12 API-only. Agent-mode systems are accessed through provider-managed CLI surfaces that provide agentic scaffolding, such as multi-turn interaction and managed execution environments. API-only systems are accessed through direct model endpoints without such scaffolding. For API-mode evaluation, Qwen, Gemini, and Claude are served through Google Cloud Vertex AI [1], while OpenAI models are accessed through the official OpenAI API [2].

Submission Format and Execution Setup. To standardize outputs, all systems must return edits as a JSON dictionary mapping file names to file contents. A local Python harness materializes each submission by overwriting the corresponding repository files. In multi-step cases, agent-mode systems receive tasks sequentially and may observe their own prior edits, whereas API-only systems receive each step independently together with the relevant prior context. In all settings, models see only the agent-facing `TASK.md`, without hidden hints or additional maintainability instructions.

Experimental Constraints. To prevent evaluator leakage, neither agent-mode nor API-only systems can access the hidden unit tests or Python evaluation scripts. Agent-mode systems are additionally restricted to read-only repository access: they may inspect files in the target repository, but cannot execute arbitrary commands, use external tools, or access files outside the workspace. These controls ensure that outputs are derived only from `TASK.md` and the visible repository contents.

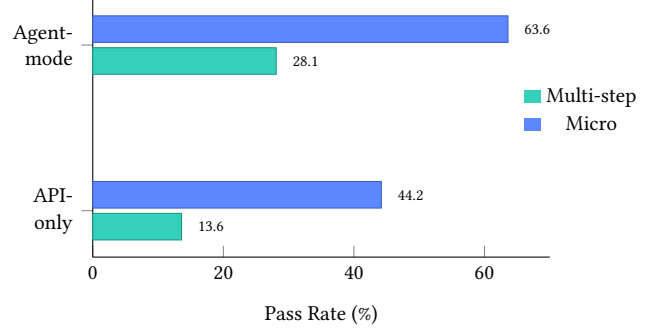


Figure 5: Both API-only and agent-mode configurations perform substantially better on micro cases than on multi-step cases, indicating that the main difficulty lies in evolutionary tasks rather than isolated edits.

5 Experimental Results and Findings

5.1 RQ1: Overall Performance

Table 3 summarizes performance across the 21-case suite. Overall pass rates range from 24% (5/21 cases) for the weakest configurations to 57% (12/21) for the strongest ones. The best-performing systems are GPT-5.3-Cx (Agt) and Claude Opus 4.6 (Agt), which each solve 12 of 21 cases. The strongest API-only configuration, Gemini 3.1 Pro (API), solves 8 of 21 cases (38%). Taken together, these results indicate that even frontier coding systems solve only about half of the NITR suite at best. Maintainability-preserving repository evolution therefore remains a substantial open challenge rather than a nearly solved capability.

Performance is highly uneven across the suite, suggesting that current systems succeed mainly when the repository already exposes a clear path for change, but struggle once they must infer and preserve that path themselves. A small number of probes are close to saturation. Case 012 (*cache lifecycle*) is solved by all 23 configurations, while Case 003 (*reuse existing code*), Case 009 (*session-expiry testability*), and Case 010 (*logging side-effects*) are passed by 21–22 systems. These are cases where the starter repository already provides a strong structural path toward the desired change, so the main burden is recognizing and following that path correctly.

At the other extreme, five probes are not solved by any configuration: Case 002 (*refactor-and-reuse*), Case 004 (*responsibility decomposition in the CV pipeline*), Case 005 (*pricing extension structure*), Case 015 (*pipeline provider decoupling*), and Case 021 (*inline filter endpoint reuse*). These universally unsolved cases are revealing because they are not simply hard programming tasks. Rather, they require the model to preserve or recover an architectural direction under structural pressure: refactor toward existing abstractions, maintain responsibility boundaries, extend behavior without patching central logic, or route changes through the repository’s intended reuse path. This suggests that the main limitation exposed by NITR is not raw implementation ability alone, but difficulty maintaining design discipline during repository-level change.

A further pattern emerges when we separate micro cases from multi-step cases, also shown in Figure 5. Across all 23 configurations, micro cases are passed at 53.5% (123/230), whereas multi-step cases

Table 2: All 21 repositories in NITR with their dimension, granularity, and aggregate outcomes across 23 evaluated configurations. S counts are bolded where ≥ 4 , highlighting probes where functionally correct code still fails the maintainability oracle.

	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015	016	017	018	019	020	021
Dim.	D1	D2	D2	D3	D4	D5	D5	D6	D7	D9	D1	D8	D1	D4	D6	D7	D8	D7	D9	D3	D2
Type	<i>m</i>	<i>m</i>	<i>m</i>	<i>m</i>	<i>m</i>	μ	<i>m</i>	μ	μ	μ	μ	μ	μ	μ	<i>m</i>	<i>m</i>	<i>m</i>	<i>m</i>	<i>m</i>	μ	μ
Pass	5	0	22	0	0	2	10	2	21	21	10	23	18	19	0	8	4	2	1	7	0
S	13	0	0	0	0	4	0	4	0	0	0	0	0	0	0	0	0	0	0	11	0

m = multi-step; μ = micro; **S** = functional tests pass, but the maintainability oracle fails.

Table 3: Pass rates of evaluated configurations on NITR (21 cases). Agent uses agentic scaffolding; API uses direct inference. Per-dimension cell background: all pass, majority, minority, none.

Model	Mode	#	Rate	D1	D2	D3	D4	D5	D6	D7	D8	D9
GPT-5.3-Cx	Agent	12	57%	2/3	1/3	1/2	1/2	1/2	1/2	2/3	2/2	1/2
GPT-5.2-Cx	Agent	11	52%	3/3	1/3	0/2	1/2	1/2	1/2	2/3	1/2	1/2
GPT-5	Agent	11	52%	2/3	1/3	1/2	1/2	1/2	0/2	3/3	1/2	1/2
GPT-5.4	Agent	10	48%	2/3	1/3	1/2	1/2	1/2	0/2	2/3	1/2	1/2
GPT-5.4	API	7	33%	1/3	1/3	0/2	1/2	0/2	0/2	1/3	2/2	1/2
GPT-5-Mini	API	6	29%	0/3	1/3	0/2	1/2	1/2	0/2	1/3	1/2	1/2
GPT-5.3-Cx	API	6	29%	0/3	1/3	0/2	1/2	1/2	0/2	1/3	1/2	1/2
Claude Opus 4.6	Agent	12	57%	3/3	1/3	1/2	1/2	2/2	0/2	2/3	1/2	1/2
Claude Opus 4.5	Agent	10	48%	2/3	1/3	1/2	1/2	1/2	0/2	2/3	1/2	1/2
Claude Sonnet 4.6	Agent	9	43%	2/3	1/3	0/2	1/2	1/2	0/2	2/3	1/2	1/2
Claude Sonnet 4.5	Agent	8	38%	2/3	1/3	1/2	0/2	1/2	0/2	2/3	1/2	0/2
Claude Opus 4.5	API	6	29%	1/3	1/3	0/2	1/2	0/2	0/2	1/3	1/2	1/2
Claude Opus 4.6	API	6	29%	1/3	1/3	0/2	1/2	0/2	0/2	1/3	1/2	1/2
Claude Sonnet 4.5	API	6	29%	1/3	1/3	0/2	1/2	0/2	0/2	1/3	1/2	1/2
Claude Sonnet 4.6	API	5	24%	0/3	1/3	0/2	1/2	0/2	0/2	1/3	1/2	1/2
Gemini 3.1 Pro	Agent	10	48%	3/3	0/3	0/2	1/2	1/2	0/2	1/3	2/2	2/2
Gemini 3.1 Flash	Agent	6	29%	1/3	1/3	0/2	1/2	0/2	0/2	1/3	1/2	1/2
Gemini 2.5 Pro	Agent	5	24%	1/3	1/3	0/2	1/2	0/2	0/2	1/3	1/2	0/2
Gemini 3.1 Pro	API	8	38%	2/3	1/3	1/2	1/2	0/2	0/2	1/3	1/2	1/2
Gemini 2.5 Pro	API	6	29%	2/3	1/3	0/2	0/2	0/2	0/2	0/3	2/2	1/2
Gemini 3.1 Flash	API	5	24%	1/3	1/3	0/2	0/2	0/2	0/2	1/3	1/2	1/2
Qwen3-Coder	API	5	24%	1/3	1/3	0/2	0/2	0/2	0/2	1/3	1/2	1/2
Qwen3-80B	API	5	24%	0/3	1/3	0/2	1/2	0/2	0/2	1/3	1/2	1/2
Agent avg.		9.5	45%									
API avg.		5.9	28%									
Case pass rate				48%	32%	15%	41%	26%	4%	45%	59%	48%

are passed at only 20.6% (52/253). The same drop appears in both evaluation regimes. API-only configurations fall from 44.2% on micro cases (53/120) to 13.6% on multi-step cases (18/132), while agent-mode configuration fall from 63.6% (70/110) to 28.1% (34/121). Moreover, four of the five universally unsolved cases belong to the multi-step setting.

Summary 1

NITR shows that current AI coding tools remain far from robust: they solve only 36.2% of cases, and pass rates collapse from 53.5% on micro cases to 20.6% on multi-step cases. The real bottleneck is sustained structural discipline under change, not isolated code generation or bug fixing.

5.2 RQ2: Which Maintainability Dimensions Are Most Challenging?

Table 3 (bottom row) reports aggregate pass rates by maintainability dimension. The results reveal a non-uniform difficulty pattern within NITR rather than a uniform decline across the suite. Please note that because some dimensions are represented by a few cases, these rates should be read as diagnostic trends within NITR rather than high-confidence population estimates of maintainability difficulty.

Dependency Control (D6) is the hardest dimension by a wide margin, with a pass rate of just 4.3%. Across its two cases (Case 008 and Case 015), only 2 of 46 evaluated attempts succeed. *Responsibility Decomposition* (D3) is the second hardest at 15.2%, followed by *Interface and Substitutability Discipline* (D5) at 26.1%, and *Reuse and Repo Awareness* (D2) at 31.9%. These are dimensions that require the model to identify and preserve latent repository structure: respect dependency boundaries, maintain responsibility separation, extend behavior through the right abstraction, and reuse existing mechanisms instead of introducing parallel paths.

By contrast, the easiest dimension is *State Ownership and Lifecycle* (D8) at 58.7%. It is followed by *Change Locality* (D1) and *Side-Effect Isolation* (D9), both at 47.8%, and *Testability and Determinism* (D7) at 44.9%. These dimensions are still nontrivial, but in many cases the intended structural direction is more explicit in the starter repository.

Summary 2

Within NITR, maintainability difficulty is highly concentrated instead of evenly distributed. Current AI coding tools achieve 58.7% on D8 but only 4.3% on D6, a 13× gap. Within NITR, the bottlenecks are dependency control, responsibility decomposition, and abstraction-respecting extension.

5.3 RQ3: Maintainability Failures Missed by Functional Correctness

To answer RQ3, we analyze whether systems fail and *how* they fail. In particular, we distinguish ordinary functional failures from failures that remain behaviorally correct yet violate the maintainability constraint encoded by the probe.

Figure 6 summarizes the outcome of every (model, case) pair. We classify each result as **Pass**, **F** (functional test failure), **S** (functional tests pass, but the maintainability oracle fails), **B** (both fail), or **E** (build error). Across all 483 evaluated pairs, 36.2% pass, 31.5% are functional failures, 13.3% are **S**-category failures, 10.4% are joint failures, and 8.7% are build errors. By this decoupling, NITR is deliberately constructed so that a behaviorally correct shortcut and a maintainable repository-aligned solution can both satisfy the immediate task, while differing in long-term structural quality.

The **S** category is especially important for answering this research question. These are solutions that compile, pass the hidden functional tests, and would be counted as successful under conventional test-based evaluation, yet still violate the intended maintainability constraint. Across the full suite, this occurs in 64

of 483 evaluated pairs (13.3%). The effect is particularly strong in Case 001, Case 002, and Case 020, which exhibit 17/23, 16/23, and 15/23 **S**-category outcomes, respectively. In these probes, many systems produce code that appears correct behaviorally while silently introducing structural debt. This shows that functional correctness and maintainability quality often decouple, and that test-only evaluation systematically overestimates maintainability competence on a nontrivial portion of the suite.

Manual Inspection. To contextualize the failure matrix, we qualitatively inspected failed outcomes at the level of individual (model, case) runs.

Across the 308 failed runs, the breakdowns clustered into a small set of recurring cross-case archetypes rather than appearing arbitrary or case-specific. These archetypes capture *empirical failure mechanisms* shared across maintainability dimensions: reuse and dependency-control probes often fail through shortcut-over-reuse or incomplete abstraction refactors, whereas responsibility and side-effect probes more often fail through boundary contamination. Overall, NITR suggests that *current AI coding tools often preserve required behavior through structurally convenient shortcuts that accumulate maintainability debt beyond what functional tests alone expose*.

Summary 3

Within NITR, test passing is an unreliable proxy for maintainability preservation. 64/483 outcomes (13.3%) are behaviorally correct yet structurally wrong, and 308 failed runs collapse into five recurring archetypes rather than case-specific noise.

5.4 RQ4: Agents vs. APIs Under the NITR Harness

Under our constrained harness, agent-mode configurations are associated with higher overall performance on NITR. We report this as an observational comparison rather than a clean causal estimate of scaffolding alone, since agent and API settings differ in interface and execution surface. Across all configurations, agent mode averages 45.0% (9.45/21 cases) versus 28.2% (5.92/21) for API-only systems, a gain of 16.8 percentage points. The best agent configuration solves 12 of 21 cases, compared with 8 for the best API-only configuration.

The same pattern appears in matched same-family comparisons. Among the nine families evaluated in both modes, agent mode improves performance in eight, with an average gain of 3.0 passed cases. For example, GPT-5.3-Cx and Claude Opus 4.6 improve from 6 to 12 passed cases in agent mode. Across NITR, the largest gains appear on Cases 007, 011, 016, and 020, which require coordination across module boundaries, staged modifications, or responsibility-preserving changes.

However, agent mode does not remove the core bottlenecks exposed by NITR: no matched family solves Cases 002, 004, 005, 015, or 021 in either mode. Overall, under this harness, agent-mode systems are associated with better maintainability-preserving performance, but they do not resolve the underlying architectural failure modes.

Model	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015	016	017	018	019	020	021
GPT-5.3-Cx (Agt)	S	S	✓	F	B	S	✓	✓	✓	✓	✓	✓	✓	✓	F	✓	✓	F	F	✓	F
GPT-5.3-Cx (API)	S	S	✓	F	B	B	✓	F	✓	✓	F	✓	F	✓	F	F	S	F	B	S	F
GPT-5.4 (Agt)	S	S	✓	F	B	S	✓	F	✓	✓	✓	✓	✓	✓	F	✓	S	F	B	✓	F
GPT-5.4 (API)	S	S	✓	F	B	E	F	F	✓	✓	F	✓	✓	✓	F	F	✓	F	B	S	F
GPT-5.2-Cx (Agt)	✓	B	✓	F	B	S	✓	✓	✓	✓	✓	✓	✓	✓	F	✓	S	F	B	S	F
GPT-5 (Agt)	S	B	✓	E	B	S	✓	B	✓	✓	✓	✓	✓	✓	F	✓	S	✓	B	✓	F
GPT-5-Mini (API)	S	B	✓	F	E	E	✓	B	✓	✓	F	✓	F	✓	F	F	F	F	B	S	F
Claude Opus 4.5 (Agt)	S	B	✓	F	F	S	✓	F	✓	✓	✓	✓	✓	✓	F	✓	S	F	B	✓	F
Claude Opus 4.5 (API)	S	S	✓	F	F	B	F	B	✓	✓	F	✓	✓	✓	F	F	B	F	B	S	F
Claude Opus 4.6 (Agt)	✓	S	✓	F	B	✓	✓	F	✓	✓	✓	✓	✓	✓	F	✓	B	F	B	✓	F
Claude Opus 4.6 (API)	S	S	✓	E	F	B	F	F	✓	✓	F	✓	✓	✓	F	F	B	F	B	S	F
Claude Sonnet 4.5 (Agt)	S	S	✓	F	B	E	✓	E	✓	F	✓	✓	✓	✓	F	✓	B	F	B	✓	F
Claude Sonnet 4.5 (API)	S	S	✓	F	E	F	B	F	✓	✓	✓	✓	✓	✓	F	F	B	F	B	S	F
Claude Sonnet 4.6 (Agt)	S	S	✓	F	F	✓	F	F	✓	✓	✓	✓	✓	✓	F	✓	B	F	B	S	F
Claude Sonnet 4.6 (API)	S	S	✓	E	E	F	B	F	✓	✓	F	✓	F	✓	F	F	S	F	B	S	F
Gemini 3.1 Pro (Agt)	✓	E	F	F	F	S	✓	S	F	✓	✓	✓	✓	✓	S	B	✓	✓	✓	S	F
Gemini 3.1 Pro (API)	S	S	✓	F	F	E	✓	S	✓	✓	✓	✓	✓	✓	F	F	S	F	B	✓	F
Gemini 3.1 Flash (Agt)	S	S	✓	F	F	F	F	F	✓	✓	F	✓	✓	✓	F	F	B	F	E	S	F
Gemini 3.1 Flash (API)	S	S	✓	E	F	E	F	F	✓	✓	F	✓	✓	F	F	F	B	F	E	S	F
Gemini 2.5 Pro (Agt)	B	E	✓	E	E	E	F	F	✓	F	✓	✓	✓	✓	F	E	E	F	E	S	F
Gemini 2.5 Pro (API)	✓	S	✓	F	E	B	E	F	E	✓	F	✓	✓	✓	F	E	✓	F	E	S	F
Qwen3-Coder (API)	✓	E	✓	E	B	E	F	E	✓	✓	F	✓	F	E	F	F	E	F	B	E	F
Qwen3-80B (API)	S	S	✓	E	E	E	F	E	✓	✓	F	✓	F	✓	E	E	B	F	B	S	F
Pass	5/23	0/23	22/23	0/23	0/23	2/23	10/23	2/23	21/23	21/23	10/23	23/23	18/23	19/23	0/23	8/23	4/23	2/23	1/23	7/23	0/23

Figure 6: Pass/fail heatmap of 23 evaluated configurations across 21 cases. ■ Pass, ■ S-category failure (functional tests pass, maintainability oracle fails), ■ Functional test failure, ■ Both fail, ■ Build failure. Cases marked ■ confirm that functional tests alone miss these maintainability failures, which are detectable only via structural probes.

Summary 4

Under the NITR harness, agent-mode configurations generally perform better than API-only ones (45.0% vs. 28.2%), but this does not isolate scaffolding as the sole cause and does not remove the core architectural bottlenecks.

5.5 Representative Case Studies

We discuss three representative cases to illustrate distinct maintainability failure modes. Cases 001 and 020 capture NITR’s central behavioral–structural gap, while Case 006 highlights brittle extension against an existing repository convention.

Case 001 (D1, S-category maintainability failure). The task asks agents to implement a multi-type add function across three incremental steps. Although the locked `app/main.cc` calls `add` with all four types from step 1, only the integer path is tested initially, so explicit overloads can still pass early. By step 3, 17 of 23 configurations incur structural debt by introducing duplicated, type-specific implementations while still passing the functional tests. The structural probe requires a generic definition (e.g., `template` or `auto`) and flags multiple explicit overloads as `ERR_DUPLICATED_IMPLEMENTATION`.

Case 020 (D3, S-category maintainability failure). The task asks agents to implement a handover-packet ownership boundary between two adjacent pipeline modules. 7 configurations pass the case. 15 of 23 configurations produce functionally correct solutions,

but the structural probe detects that the ownership-transfer function also performs validation or enrichment that belongs in the receiving module, violating the intended responsibility boundary.

Case 006 (D5, mixed maintainability/functional/build failure). The task asks agents to refactor a hit-processing pipeline so that the sort stage consumes a narrow *view* type rather than the full `HitBuffer`. Two configurations pass. 6 configurations still pass all functional tests, but introduce no new view type. Here the repository already establishes an extension contract through its existing type-family structure, and the task requires the new sort-stage interface to follow that pattern. The structural probe therefore checks for a corresponding view abstraction in the public interface using repository-consistent patterns (e.g., `SortView` or `IHitSorting`); this is not a stylistic naming preference, but a check that the edit preserves the repository’s extension structure rather than taking an ad hoc shortcut. A further four configurations introduce a view type but wire it incorrectly, breaking functionality, and eight fail to compile altogether.

6 Discussion and Implications

6.1 Implications

NITR suggests implications for *education*, *code review*, and *agent design*. First, NITR can be useful as a teaching instrument for SE courses. Its cases make design tradeoffs concrete by contrasting behaviorally correct solutions with structurally poor ones, which help students discuss reuse, boundaries, and testability in a more

Table 4: The table summarizes recurring qualitative patterns and representative cases; it is not an exhaustive coded annotation of all failures.

Failure archetype	Defining behavior	Representative cases	Outcome
Shortcut over reuse or abstraction	Clones logic or creates a parallel path instead of routing the change through a reusable or repository-aligned abstraction	001, 002, 011, 021	S / F
Boundary contamination	Preserves visible behavior but violates responsibility, ownership, or side-effect boundaries	004, 019, 020	S / F / B
Incomplete abstraction refactor	Starts the right abstraction shape but does not propagate the change consistently across the repository	006, 008, 014, 015	E / B
Later-step regression	Works for earlier milestones but breaks baseline behavior or collapses under later extensions	005, 007, 016, 017, 018, 019	F / B
Missing test seam	Leaves time, randomness, or configuration-sensitive behavior hardwired in core logic	009, 016, 018	F

grounded way. Second, in practice, reviewing AI-generated code should go beyond asking whether a patch works, and ask whether it preserves reuse paths, dependency boundaries, responsibility separation, etc. Especially for multi-step changes, AI-generated code should be treated less as finished output and more as a structurally untrusted proposal. Third, the failures exposed by NITR suggest that improving coding agents will require stronger repository-level structural reasoning rather than better local code synthesis alone.

6.2 Threats to Validity

Construct validity. NITR operationalizes maintainability through authored probes and case-specific structural oracles. This means the framework measures maintainability preservation as instantiated by explicit repository-evolution pressures and executable design-boundary checks, not maintainability in an unrestricted sense. We therefore position NITR as a diagnostic instrument for targeted maintainability pressures rather than a universal maintainability benchmark. **Internal validity.** Agent/API comparisons depend on the execution surfaces used here and should be read as harness-level comparisons rather than isolated causal estimates. Each configuration is evaluated with a fixed setup and single sampled run per case, so we do not estimate run-to-run variance or prompt sensitivity. **External validity.** The suite uses 21 small C++ repositories for diagnostic clarity. This supports interpretability and oracle precision, but does not cover the full diversity of languages, repository scales, or software-engineering workflows. **Conclusion validity.** Some maintainability dimensions are represented by only two or three cases. Accordingly, dimension-level differences are best interpreted as suite-level diagnostic trends rather than high-power statistical estimates.

7 Related Work

Software Engineering Benchmarks for LLMs. A large body of work evaluates coding systems primarily through behavioral correctness. Early benchmarks such as HumanEval [10], MBPP [5], and CodeContests [26] focus on synthesis tasks whose outputs are judged by hidden tests. More recent benchmarks broaden the

setting from isolated functions to realistic codebase tasks while retaining correctness as the main signal. RepoBench [27] introduces repository-level context for cross-file completion, LiveCodeBench [20] emphasizes contamination-free and continually refreshed evaluation, SWE-bench [21] measures issue resolution in real GitHub repositories, and EvoCodeBench [25] evaluates repository-grounded code generation under evolving benchmark updates.

This line of work has since expanded along several axes. SWE-bench Verified [12], SWE-bench Pro [15], and Multi-SWE-bench [41] strengthen issue-resolution evaluation through improved quality control, difficulty, and multilingual coverage. SWE-Evo [23] and Commit-0 [44] move toward longer-horizon development workflows, while NL2Repo [16] studies full repository generation from natural-language specifications. Automated pipelines such as SWE-rebench [6] and SWE-bench Live [42] further improve realism.

These works are indispensable for measuring task-completion capability, but their primary signal remains test behavior. NITR instead evaluates whether a behaviorally correct repository edit preserves maintainability. It relies on expert-curated probes to ensure data quality, so that each case cleanly encodes a specific software engineering principle and maintainability pressure.

Long-Horizon Software Evolution Benchmarks. A second line of work pushes evaluation beyond one-shot issue resolution toward broader software-development behavior and continuous repository change. DevBench [24] evaluates multiple stages of the software-development lifecycle, SWE-EVO [23] and SWE-CI [9] study multi-step repository evolution and continuous-integration-style maintenance, SlopCodeBench [3] examines code erosion as agents iterate, and EvoClaw [14] evaluates continuous software evolution through executable milestone sequences that stress sustained system integrity over time.

These efforts primarily evaluate agents under continuous task evolution measured by test behaviors, whereas NITR evaluates whether a behaviorally correct repository edit preserves maintainable structure in a way that can be diagnosed precisely.

LLMs for Design Quality and Maintainability Analysis. Maintainability has long been studied in software engineering through

code and design quality perspectives, including coupling and cohesion metrics [11], design principles such as SOLID [28], and technical debt as a way to reason about future maintenance cost [13].

Recent software engineering work has begun to study LLMs on maintainability and quality-oriented tasks beyond functional correctness [29, 35]. Recent examples include LLM-based localization of code design issues [7], prompting-based detection of SOLID principle violations [32], evaluation of LLMs for fixing maintainability issues in real-world projects [30], context-enriched quality-aware code review generation [43], and industrial experience with LLM-based automated code review [33].

These efforts study how LLMs can *analyze*, *review*, or *repair* maintainability-related problems. NITR differs from them by evaluating whether maintainability is preserved during code generation itself. Its distinguishing mechanism is the use of hidden structural oracles tied to explicit repository-evolution pressures.

8 Conclusion

NITR is ultimately a step toward a broader shift in how we evaluate LLM for software engineering: from asking whether models can produce correct code, to asking whether they can participate responsibly in the long-term evolution of real repositories. As coding agents move from isolated completions to sustained collaboration with developers, the central question is no longer only correctness, but whether generated changes preserve the structural conditions that make future change possible. We hope NITR helps make maintainability a first-class objective in both evaluation and system design, and encourages the next generation of coding agents to be judged not only by what they can build today, but by whether what they build remains worth extending tomorrow.

Acknowledgments

The authors of this work are in part supported by NSF CCF-2426161, NSF CAREER Award IIS-2541224, Google Research Credit Awards, and Startup Funding offered by Tulane University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Data Availability Statement

We provide the open source artifact at: <https://github.com/ucrriple/NITR>. A long-term accessible version with an assigned DOI is archived at <https://doi.org/10.5281/zenodo.21154293> [46].

References

- [1] 2026. Google Vertex AI. <https://cloud.google.com/vertex-ai>.
- [2] 2026. OpenAI API Platform. <https://openai.com/api/>.
- [3] 2026. SlopCodeBench. <https://www.scbench.ai/>.
- [4] 2026. Why the majority of vibe coded projects fail. https://www.reddit.com/r/ClaudeAI/comments/1rt31th/why_the_majority_of_vibe_coded_projects_fail/.
- [5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021). doi:10.48550/arXiv.2108.07732
- [6] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. 2025. SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents. arXiv:2505.20411 [cs.SE] <https://arxiv.org/abs/2505.20411>
- [7] Fraol Batole, David O'Brien, Tien N. Nguyen, Robert Dyer, and Hridesh Rajan. 2025. An LLM-Based Agent-Oriented Approach for Automated Code Design Issue Localization. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) (ICSE '25). IEEE Press, 1320–1332. doi:10.1109/ICSE55347.2025.00100
- [8] Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering* (ICSE). 2188–2200. <https://arxiv.org/abs/2403.17134>
- [9] Jialong Chen, Xander Xu, Hu Wei, Chuan Chen, and Bing Zhao. 2026. SWE-CI: Evaluating Agent Capabilities in Maintaining Codebases via Continuous Integration. *arXiv preprint arXiv:2603.03823* (2026). doi:10.48550/arXiv.2603.03823
- [10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgun Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021). doi:10.48550/arXiv.2107.03374
- [11] Shyam R. Chidamber and Chris F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493. doi:10.1109/32.295895
- [12] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubei, Mia Glaese, et al. 2024. Introducing SWE-bench Verified. <https://openai.com/index/introducing-swe-bench-verified/>.
- [13] Ward Cunningham. 1992. The WyCash Portfolio Management System. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA '92). ACM, 29–30. doi:10.1145/157709.157715
- [14] Gangda Deng, Zhaoling Chen, Zhongming Yu, Haoyang Fan, Yuhong Liu, Yuxin Yang, Dhruv Parikh, Rajgopal Kannan, Le Cong, Mengdi Wang, Qian Zhang, Viktor Prasanna, Xiangru Tang, and Xingyao Wang. 2026. EvoClaw: Evaluating AI Agents on Continuous Software Evolution. arXiv:2603.13428 [cs.SE] <https://arxiv.org/abs/2603.13428>
- [15] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. 2025. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks? arXiv:2509.16941 [cs.SE] <https://arxiv.org/abs/2509.16941>
- [16] Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, Weiran Shi, Zaiyuan Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Qizhi Chen, Xianfu Cheng, Bo Deng, Qingshui Gu, Kai Hua, Juntao Lin, Pai Liu, Mingchen Li, Xuan-guang Pan, Zifan Peng, Yujia Qin, Yong Shan, Zhewen Tan, Weihao Xie, Zihan Wang, Yishuo Yuan, Jiayu Zhang, Enduo Zhao, Yunfei Zhao, He Zhu, Liya Zhu, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyang Tao, Jian Yang, Tong Yang, Zhaoliang Zhang, Xinjie Chen, Wenhao Huang, and Ge Zhang. 2026. NL2Repo-Bench: Towards Long-Horizon Repository Generation Evaluation of Coding Agents. arXiv:2512.12730 [cs.CL] <https://arxiv.org/abs/2512.12730>
- [17] Roy S. Freedman. 1991. Testability of Software Components. *IEEE Trans. Softw. Eng.* 17, 6 (June 1991), 553–564. doi:10.1109/32.87281
- [18] Stefan Haeffliger, Georg Von Krogh, and Sebastian Spaeth. 2008. Code reuse in open source software. *Management science* 54, 1 (2008), 180–193. doi:10.1287/mnsc.1070.0748
- [19] Ruida Hu, Chao Peng, Jingyi Ren, Bo Jiang, Xiangxin Meng, Qinyun Wu, Pengfei Gao, Xinchun Wang, and Cuiyun Gao. 2024. A Real-World Benchmark for Evaluating Fine-Grained Issue Solving Capabilities of Large Language Models. *arXiv preprint arXiv:2411.18019* (2024). <https://arxiv.org/abs/2411.18019>
- [20] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanfai Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Live-CodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *arXiv preprint arXiv:2403.07974* (2024). doi:10.48550/arXiv.2403.07974
- [21] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=VTF8yNQM66>
- [22] Charles W. Krueger. 1992. Software reuse. *ACM Comput. Surv.* 24, 2 (June 1992), 131–183. doi:10.1145/130844.130856

- [23] Tue Le, Minh V. T. Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. 2026. SWE-EVO: Benchmarking Coding Agents in Long-Horizon Software Evolution Scenarios. *arXiv:2512.18470* [cs.SE] <https://arxiv.org/abs/2512.18470>
- [24] Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, Zhiyin Yu, He Du, Ping Yang, Dahua Lin, Chao Peng, and Kai Chen. 2024. DevBench: A Comprehensive Benchmark for Software Development. *arXiv preprint arXiv:2403.08604* (2024). doi:10.48550/arXiv.2403.08604
- [25] Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024. EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories. *arXiv preprint arXiv:2404.00599* (2024). doi:10.48550/arXiv.2404.00599
- [26] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustín Del Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Audume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *arXiv preprint arXiv:2203.07814* (2022). doi:10.48550/arXiv.2203.07814
- [27] Tianyang Liu, Canwen Xu, and Julian McAuley. 2024. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. In *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=pPjZIOuQuF>
- [28] Robert C. Martin. 2003. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall.
- [29] Alfred Santa Molison, Marcia Moraes, Glaucia Melo, Fabio Santos, and Wesley K. G. Assuncao. 2025. Is LLM-Generated Code More Maintainable & Reliable than Human-Written Code? *arXiv preprint arXiv:2508.00700* (2025). doi:10.48550/arXiv.2508.00700
- [30] Henrique Nunes, Eduardo Figueiredo, Larissa Rocha, Sarah Nadi, Fischer Ferreira, and Geanderson Esteves. 2025. Evaluating the Effectiveness of LLMs in Fixing Maintainability Issues in Real-World Projects. *arXiv:2502.02368* [cs.SE] <https://arxiv.org/abs/2502.02368>
- [31] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph. In *The Thirteenth International Conference on Learning Representations (ICLR)*. <https://openreview.net/forum?id=JbWVgWsW9H>
- [32] Fatih Pehlivan, Arçın Ülkü Ergüzen, Sahand Moslemi Yengejeh, Mayasah Lami, and Anil Koyuncu. 2025. Are We SOLID Yet? An Empirical Study on Prompting LLMs to Detect Design Principle Violations. *arXiv:2509.03093* [cs.SE] <https://arxiv.org/abs/2509.03093>
- [33] Shweta Ramesh, Joy Bose, Hamender Singh, A K Raghavan, Sujoy Roy Chowdhury, Giriprasad Sridhara, Nishrith Saini, and Ricardo Britto. 2025. Automated Code Review Using Large Language Models at Ericsson: An Experience Report. *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2025), 602–607. <https://api.semanticscholar.org/CorpusID:280046022>
- [34] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R. Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems*. <https://openreview.net/forum?id=vAElhFCKW6>
- [35] Xin Sun, Daniel Ståhl, Kristian Sandahl, and Christoph Kessler. 2025. Quality Assurance of LLM-generated Code: Addressing Non-Functional Quality Characteristics. *arXiv preprint arXiv:2511.10271* (2025). doi:10.48550/arXiv.2511.10271
- [36] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*. 50208–50232. <https://arxiv.org/abs/2402.01030>
- [37] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *The Thirteenth International Conference on Learning Representations (ICLR)*. <https://openreview.net/forum?id=Ojd3ayDDoF>
- [38] T.W. Williams and K.P. Parker. 1983. Design for testability—A survey. *Proc. IEEE* 71, 1 (1983), 98–112. doi:10.1109/PROC.1983.12531
- [39] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems*. <https://openreview.net/forum?id=mXpq6ut8J3>
- [40] John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. 2025. SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains?. In *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=riTiq3i21b>
- [41] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. 2025. Multi-SWE-bench: A Multilingual Benchmark for Issue Resolving. *arXiv:2504.02605* [cs.SE] <https://arxiv.org/abs/2504.02605>
- [42] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2025. SWE-bench Goes Live! *arXiv:2505.23419* [cs.SE] <https://arxiv.org/abs/2505.23419>
- [43] Yuxin Zhang, Yuxia Zhang, Zeyu Sun, Yanjie Jiang, and Hui Liu. 2026. LAURA: Enhancing Code Review Generation with Context-Enriched Retrieval-Augmented LLM. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Seoul, Korea, Republic of). IEEE Press, 2983–2995. doi:10.1109/ASE63991.2025.00245
- [44] Wenting Zhao, Nan Jiang, Celine Lee, Justin T. Chiu, Claire Cardie, Matthias Gallé, and Alexander M. Rush. 2025. Commit0: Library Generation from Scratch. In *International Conference on Learning Representations (ICLR)*.
- [45] Jiasheng Zheng, Boxi Cao, Zhengzhao Ma, Ruotong Pan, Hongyu Lin, Yaojie Lu, Xianpei Han, and Le Sun. 2024. Beyond Correctness: Benchmarking Multi-dimensional Code Generation for Large Language Models. *arXiv preprint arXiv:2407.11470* (2024). <https://arxiv.org/abs/2407.11470>
- [46] Haichao Zhu, Qian Zhang, Jiyuan Wang, Zhaorui Yang, and Yuxin Qiu. 2026. *Artifacts for Diagnosing Maintainability Failures*. doi:10.5281/zenodo.21154293

Received 2026-03-26; accepted 2026-06-18