SPECALIGN: TEST-DRIVEN SPECIFICATION REFINEMENT FOR CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Software specifications, such as issue descriptions, are usually informal and often vague. Nevertheless, users insist that code written or generated based on a specification must align with it. This may require clarifying the specification itself. Prior work attempted to improve code+spec alignment by automatically rewriting the specification in a standardized way. Unfortunately, doing so just shifts the problem: now it is burdensome for users to tell whether the standardized spec aligns with the original. Instead, we propose SPECALIGN, which takes a different approach: it pinpoints parts of the spec to improve and surgically rewrites only those parts. That makes it is easier to compare the original and rewritten spec. To align a spec with generated code, SPECALIGN generates both code and tests, and then optimizes how well they agree. We demonstrate that this leads to better spec+code alignment, and ultimately, better issue resolution rates on SWE-Bench Pro and SWE-rebench. Even without a human, just pinpointing the gap already yields 2.4-3.6% improvement while only touching 35% of words. To understand the remaining gap, we introduce a simulated user into the SPECALIGN loop. Thanks to SPECALIGN pinpointing the gap, a simulated human can get 13.8-34.4% improvement. We quantify how often it requires information missing from the spec and available only in the ground truth. Overall, we hope that this paper contributes to better aligning AI-generated code with human intentions.

1 INTRODUCTION

A software specification describes desired behavior to guide the implementation of new or modified code. In practice, specifications, or *specs*, tend to be informal natural language, such as issue descriptions on code repositories (Jimenez et al., 2024). To correctly implement a spec, code must *align* with it, meaning its behavior must be consistent with the spec. However, the informal nature of specs makes such alignment difficult to achieve or even measure automatically. This lack of reliable signal limits the performance of AI-based coding assistants. The two most common approaches from prior work to improve code+spec alignment use either spec standardization (e.g. (Kuang et al., 2026)) or a human in the loop (HITL, e.g. (Takerngsaksiri et al., 2025)). Unfortunately, automatically standardized specs may diverge from the original in non-obvious ways, and human effort is a scarce resource that should not be requested lightly.

This paper introduces SPECALIGN, a workflow for minimally refining a spec to achieve better spec+code alignment. The core idea is to take a test-driven approach. SPECALIGN first independently generates both tests and code from the same spec. Then, it prompts an LLM to minimally update the spec based on observing how the tests did on the code. In other words, it uses agreement between tests and code as a surrogate signal for the hard-to-measure spec+code alignment. Finally, it iterates until the tests and code either agree or until a loop limit. This iterative workflow mirrors a natural software-development process: implementation and testing expose gaps in a specification, which are then clarified before the implementation is revised. Experimental results show that our test-driven approach is indeed an effective proxy for achieving spec alignment. On SWE-Bench Pro (Deng et al., 2026), SPECALIGN lifts issue resolution rate by 2.4%-3.6% depending on the model. At the same time, it keeps spec changes to only 35% of the original spec size. Results were similar on a post-model-cutoff split of SWE-rebench (Badertdinov et al., 2025).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Like all systems that have no human in the loop, SPECALIGN is limited by the fundamental ambiguity of informal issue descriptions. How much better could a human be at aligning specs? And what would human-directed specification changes look like? To answer these questions, we performed a simulated HITL study. Specifically, we extended the SPECALIGN loop with a simulated user who has access to oracular knowledge of the ground truth. This simulated user limits their effort to the small spec changes from SPECALIGN, refining them further to better align the spec. Experimental results reveal that the simulated human often makes edits for clarifications/precision refinement or to correct scope misspecifications. Despite having oracular knowledge, for 72% of instances, the HITL addition does not leak information from the ground truth; on the other hand, leakage is severe for 11% of instances. In the end, the HITL leads to 34% and 73% higher issue resolution rates on SWE-Bench Pro and SWE-rebench.

This paper makes the following novel contributions:

- SPECALIGN, a workflow for pinpointing gaps, ambiguities, and inconsistencies in software issue specifications. By analyzing disagreements between generated artifacts, SPECALIGN identifies specification regions most likely responsible for downstream failures.
- Using agreement between code and test as the measurable signal to drive alignment of the specification from which both are generated. Measuring agreement between code and tests enables SPECALIGN to precisely target specification refinements.
- An empirical study of how specification problems pinpointed by SPECALIGN could be mitigated by human developers. Our analysis characterizes the types of specification corrections developers can make and shows that improved specifications lead to better downstream code generation.

Overall, SPECALIGN helps better align AI-generated code with human intentions without demanding too much human effort.

2 RELATED WORK

Rewriting and Refining Specifications. Given the ability of LLMs and AI coding agents to generate code from natural language, specification-driven development has emerged as a promising paradigm for software engineering (Liu et al., 2026; Ruan et al., 2025). Consequently, refining specifications to improve code generation has gained attention. For isolated single-function benchmarks (e.g., MBPP and HumanEval), Interactive Test-Driven Code Generation (ITDCG) (Lahiri et al., 2023) leverages lightweight user feedback to jointly formalize the user intent as tests (a partial specification), and generates code that meets the formal user intent, improving baseline resolution rates by over 50%. At the repository level, recent works recognize that issue descriptions often lack critical context or contain ambiguities that hinder patch generation. To address this, REAgent (Kuang et al., 2026) constructs structured task specifications to guide LLM agents, reporting 15% to 35% gains across subsets of SWE-Bench, SWE-Bench Verified, and SWE-Bench Pro. Similarly, CodeScout (Suri et al., 2026) leverages codebase pre-exploration to systematically convert ambiguous user requests into refined problem statements, yielding a 20% improvement on SWE-Bench Verified. However, these techniques rely on globally rewriting or standardizing the entire specification. As a result, they shift the burden to the user, who must now validate whether the rewritten specification faithfully aligns with the original intent. In contrast, SPECALIGN pinpoints and rewrites only the problematic parts of a specification while optimizing code and test agreement.

Repository-level issue resolution and HITL. Automating repository-level issue resolution has gained momentum with the rise of agents for software engineering (SWE) (Jimenez et al., 2024; Yang et al., 2024). To improve resolution accuracy, SpecRover (Ruan et al., 2025) combines code search with iterative specification inference outperforming AutoCodeRover (Zhang et al., 2024) on SWE-Bench Lite. Concurrently with our work, empirical studies on bug report quality (Khatib et al., 2026b) revealed that explicit fix suggestions, reproduction scripts, repository source code, and localization information are associated with higher resolution likelihood, while verbose reports degrade performance. Complementing these findings, human-in-the-loop frameworks such as HULA (Takerngsaksiri et al., 2025), enable developers to interactively guide agent planning and code generation, demonstrating the value of iterative user feedback. Our work bridges these paradigms: SPECALIGN identifies specific sections of an issue description that require refinement to optimize spec alignment, and obtains developer's feedback to further enhance the quality of bug reports.

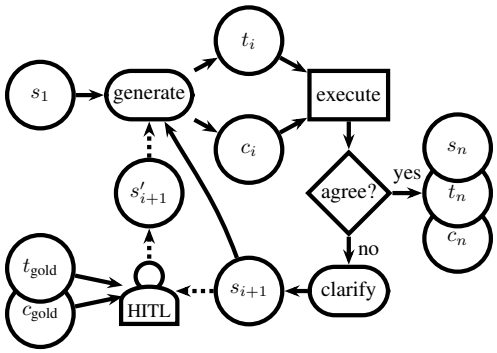


Figure 1: Specification alignment workflow. Dashed arrows: optional human-in-the-loop extension.

The store should use a small struct named `cacheEntry` (not `cacheItem`) containing both the credential and its expiry time. The cache map should be typed as `map[string]*cacheEntry`. All access to the cache must be guarded by the mutex to ensure thread safety under concurrent requests.

(a) Edit s_2 proposed by SPECALIGN

The store should use a small struct named ~~`cacheEntry`~~ ~~(not `cacheItem`)~~ `cacheItem` containing both the credential (field `credential auth.Credential`) and its expiry ~~time~~ time (field `expiresAt time.Time`). The cache map should be typed as ~~`map[string]*cacheEntry`~~ `map[string]cacheItem` (value type, not pointer). All access to the cache must be guarded by the mutex to ensure thread safety under concurrent requests.

(b) Updated edit s'_2 with SPECALIGN + HITL

Figure 2: Edits proposed by SPECALIGN for example instance_flipt-io__flipt-96820c3ad10b0b2305.

3 METHODOLOGY

This section describes SPECALIGN as well as the HITL extension used for oracular experiments.

3.1 SPECALIGN WORKFLOW

Fig. 1 shows the overall workflow of SPECALIGN. The input is an initial specification s_1 and the output is an aligned triple $\langle s_n, t_n, c_n \rangle$ of a final specification s_n, test patch t_n, and code patch c_n. A test patch may contain multiple tests, and we ensure it contains at least one regression test and one test that reproduces the issue described in the spec. A code patch may contain multiple changes to the repository that attempt to implement the specification. Each SPECALIGN iteration i starts by independently generating a test patch $t_i \sim \pi_{\text{genTest}}(s_i)$ and a code patch $c_i \sim \pi_{\text{genCode}}(s_i)$ from the current spec s_i. The details of the generators π_{genTest} and π_{genCode} are given below. To reduce the risk of test overfitting (Ahmed et al., 2026), SPECALIGN does not expose the generated code patch to the test generator nor the generated test patch to the code generator.

We expect that if the specification is well written, the generated code patch can be validated using the generated test patch. Even if the code patch cannot be validated, test execution logs can help identify gaps or ambiguities in the specification. Executing the test patch on the code patch yields $\text{logs}_i = \text{execute}(t_i, c_i)$. We primarily use the SWE-Bench Pro harness for test execution. However, SWE-Bench Pro only runs tests at file-level granularity, so we updated it to support running specific tests across all programming languages. We execute t_i before and after applying the code patch c_i. On the buggy or initial version, the reproduction test should fail, while the regression tests should pass. After applying the code patch, the reproduction test should pass, confirming that the specification has been addressed. The regression tests should continue to pass, confirming that the new code patch has not broken any existing functionality. If all of these conditions hold, the 'agree?' check in Fig. 1 succeeds and the loop terminates. Otherwise, we provide the specification, generated code and test patches, and execution logs from both the initial and patched repositories to an LLM and ask it to update the specification, producing $s_{i+1} \sim \pi_{\text{clarify}}(s_i, t_i, c_i, \text{logs}_i)$. We instruct the LLM to highlight changes using `<new>...</new>` tags. SPECALIGN then repeats the same pro-

cess from s_{i+1}. It stops after attempting to validate s_3, as further iterations tend to produce smaller spec changes with diminishing chances of improving code+test agreement while increasing cost.

Fig. 2a shows an example of an updated specification. The original specification s_1 underspecifies the structure name. SPECALIGN notices that because the generated tests t_1 and code c_1 disagree on that name, and adds an explicit structure name "cacheEntry" to the spec s_2 to improve alignment. Absent further information, this is a reasonable call, though one that ultimately conflicts with the hidden ground truth. Keeping this spec change small and localized makes it easier for a human to review, and later in this section, we will discuss how a simulated human edits this further.

3.2 TEST AND PATCH GENERATION

SPECALIGN can be coupled with any test and code generator. Here, we describe the specific generators we chose and how we adapted them to our setting.

Test Generator π_{genTest}. A *reproduction test* fails before, and passes after, the spec is correctly implemented. Reproduction test generation is an active research area, with several existing approaches, including AEGIS (Wang et al., 2024), OTTER (Ahmed et al., 2025b), AssertFlip (Khatib et al., 2026a), and iCore (Wang et al., 2026). Most test generators depend on test execution. Since execution is already a separate step in the SPECALIGN pipeline, we choose OTTER, an execution-free and lightweight test generator. OTTER consists of three major components: localization, contextualization, and test generation. It was initially designed for Python; we extend it to also support Go, JavaScript, and TypeScript, which requires language-specific preprocessing and prompt updates. A *regression test* ensures that existing functionality still works correctly after implementing the spec. Besides reproduction tests, SPECALIGN also leverages regression tests to prevent coding agents from breaking the code. We follow a simple approach to select these regression tests. During OTTER's localization step, we also identify existing tests that are relevant to the issue but do not directly address it, avoiding conflicts with the generated reproduction tests. To achieve test success, the code patch should transform the failing reproduction test into a passing one while keeping the selected regression tests passing.

Code Generator π_{genCode}. Several coding agents have been developed to resolve software issues and implement task specifications, including SWE-Agent (Yang et al., 2024), SpecRover (Ruan et al., 2025), OpenHands (Wang et al., 2025b), and Trae-Agent (Gao et al., 2025). We use mini-SWE-agent as our patch generator, a lightweight and simplified successor to SWE-Agent. However, the default mini-SWE-agent is not configured for SWE-Bench Pro, so we modified it to support this benchmark. There are growing concerns that frontier models may use Git history or other repository information to access the correct solution. To prevent such leakage, we apply several safeguards. We remove metadata that could reveal the correct solution and restrict the repository to the base version. We also prevent access to future commits and gold test files. In addition, sensitive setup and execution logs are kept hidden from the agent. These safeguards ensure that the agent generates patches using only the information provided for the target task.

3.3 SPECALIGN + HITL

This paper includes a study with a simulated user to better understand how specification problems identified by SPECALIGN could be mitigated by human developers. Fig. 2a presents an example specification update s_2 generated by SPECALIGN, with changes explicitly highlighted for the developer. A developer can review these highlighted parts and make further edits, producing the refined specification s_2' shown in Fig. 2b. In this example, SPECALIGN highlighted the need to specify a name. While the developer agrees that a name is needed, they disagree on the proposed name based on their context and preferences. Conducting this study with human developers is challenging for two main reasons. First, not all human subjects have sufficient background knowledge of the target projects. Second, identifying developers with the required expertise and involving them in a large-scale evaluation is not scalable. Since our goal is to evaluate how mitigating specification problems affects coding-agent performance, we simulate the developer using an LLM with access to the gold code and test patches, along with the model-generated code patch, test patch, and execution logs. The LLM generates an updated spec $s'_{i+1} \sim \pi_{\text{hitl}}(s_{i+1}, t_{\text{gold}}, c_{\text{gold}}, t_i, c_i, \text{logs}_i)$. It is is instructed to

modify only the highlighted portions of s_{i+1} to correct misleading or hallucinated information. As shown in the dashed arrows of Fig. 1, SPECALIGN + HITL then resumes from s'_{i+1} instead of s_{i+1}.

Because the simulated developer has access to the gold patches, there is a risk that information about the solution may leak into the refined specification, for example, through implementation suggestions or pseudocode. Section 4.4 studies the impact of such leakage on coding-agent performance, and Section 4.5 categorizes the changes made by the simulated developer to understand what types of modifications may be required from developers during specification refinement.

3.4 PATCH SELECTION

The SPECALIGN workflow in Fig. 1 terminates after at most three iterations, producing up to three triples $\langle s_i, t_i, c_i \rangle$ for $i \in \{1, 2, 3\}$. But to evaluate the quality of generated code patches, we must pick a single patch c_i to report pass@1 performance. We report results using two selection criteria.

Test-based selection. This approach evaluates each generated patch by comparing the test results before and after applying the patch. Patches receive higher scores when they turn failing tests into passing tests without causing previously passing tests to fail ($f2p > 0$ and $p2f == 0$). Patches that make only partial improvements, such as increasing the total number of passing tests, receive lower scores. If multiple versions receive the same score, the earlier version is selected. When no human is involved, we find this metric to be effective. Tests can provide a meaningful signal for SPECALIGN, as the spec may drift away from the original intent with each LLM-based modification.

Final-patch selection. This is a relatively simple selection strategy in which we choose the patch generated from the latest available specification. For example, if an instance has three versions of the specification ($s1$, $s2$, and $s3$), we select the code patch c_3 generated from $s3$. This strategy is not very helpful for SPECALIGN without human involvement. However, it can be effective in the HITL setting, where a developer provides hints in each iteration. These hints can guide the generated patch toward the intended solution even when specification alignment is not fully achieved.

4 EVALUATION

4.1 EXPERIMENTAL SETUP

Most of our experiments use SWE-Bench Pro (Deng et al., 2026), a multilingual benchmark for long-horizon software engineering tasks. It contains 731 instances across Python, Go, JavaScript, and TypeScript. We evaluate two LLMs, Claude Opus 4.6 and GPT-5.4. We observe better performance with Claude-Opus-4.6 and therefore primarily use it for subsequent analysis experiments and ablation studies. For mini-SWE-agent, we allocate a budget of $3 per iteration or a maximum of 250 steps, whichever is reached first. We use the default model configuration without higher reasoning settings. We also experiment with 200 Python instances from SWE-rebench (Badertdinov et al., 2025) that were created after the knowledge cutoff of Claude-Opus-4.6. This allows us to evaluate how SPECALIGN performs on a contamination-free benchmark.

4.2 PERFORMANCE OF CODING AGENTS USING SPECALIGN

Table 1: Performance of SPECALIGN on SWE-Bench Pro (with and without HITL)

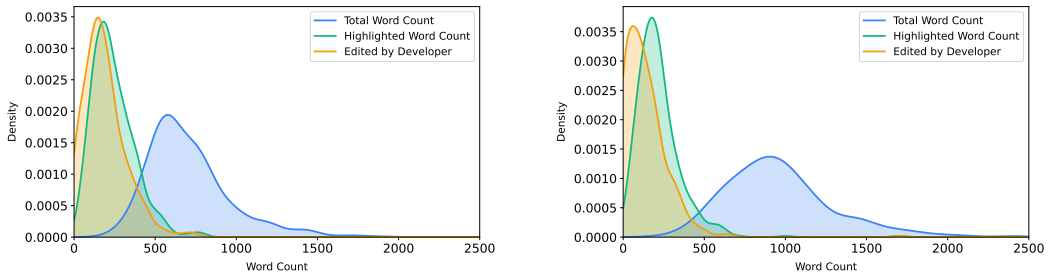
Approach	Selection	Claude-Opus-4.6					GPT-5.4				
		Resolution		Agreement (In %)	Resolved | Agreement		Resolution		Agreement (In %)	Resolved | Agreement	
		In %	Change (In %)		In %	Change (In %)	In %	Change (In %)		In %	Change (In %)
mini-SWE-agent	NA	58.1	NA	NA	NA	NA	45.6	NA	NA	NA	NA
SPECALIGN	Test-based	59.5	+2.4	40.0	66.8	+36.0%	47.2	+3.6	22.3	62.0	+36.0%
	Final-patch	59.1	+1.7				45.7	+0.3			
SPECALIGN + HITL	Test-based	64.2	+10.5	37.5	76.3	+55.4%	47.3	+3.9	16.7	65.6	+43.9%
	Final-patch	78.1	+34.4				51.8	+13.8			

Table 1 shows the impact of SPECALIGN (with and without HITL) on SWE-Bench Pro. The two 'Resolution' columns (one under each LLM) indicate the issue resolution rate across all instances

of the benchmark. SPECALIGN with test-based selection achieves a 2.4% to 3.6% relative improvement over the vanilla mini-SWE-agent. This shows that allowing LLMs to update specifications based on test execution can improve performance, although the improvement is relatively small. SPECALIGN is effective at identifying gaps in specifications but is less effective at proposing solutions to address them. However, when a human with relevant background knowledge is involved in addressing these specification gaps, the performance of mini-SWE-agent improves significantly. SPECALIGN with final-patch selection achievs a 13.8% to 34.4% relative improvement over the vanilla mini-SWE-agent. Table 7 and Table 8 in the Appendix show the performance breakdown of SPECALIGN by programming language for the test-based and final-patch selection criteria.

The two 'Agreement' columns (one under each LLM) indicate the percentage of instances for which the 'agree?' check in the SPECALIGN workflow succeeds. Even with HITL, this metric did not substantially increase. Fortunately, even when the 'agree?' check does not succeed, the gaps identified through test execution remain useful. The 'Resolved | Agreement' column shows the resolution rate restricted to just the agreeing instances; it is much higher than among all instances. This indicates that achieving agreement is, on average, beneficial. Nevertheless, failures can still occur even after agreement, potentially due to weaknesses in generated tests compared to hidden ground-truth tests.

4.3 SIZE OF EDITS MADE BY SPECALIGN



(a) s_2 and s'_2 (b) s_3 and s'_3

Figure 3: Distribution of total and edited word counts

How large are the spec changes that a human developer has to read, and how large are the additional changes they would want to write? To explore these questions, we counted words in specifications generated by the Claude-Opus-4.6 SWE-Bench Pro runs. (We counted words because tokens would be too LLM-specific and less meaningful to humans, and lines would be too coarse-grained and formatting-specific.) Fig. 3 shows word count distribution histograms for the revised specs s_i, for $i \in \{2, 3\}$. Each subfigure visualizes three distributions: 'Total Word Count' in s_i shown in blue; 'Highlighted Word Count' in s_i, i.e., words between `<new>...</new>` tags, in green; and 'Edited by Developer' in s'_i, in gold. Counting edits is challenging because a minor update to a word (e.g., "issues"→"issue") can result in a count of two if we simply sum the numbers of deleted and added words. To avoid such double-counting, we divide the changes into hunks. A hunk is a contiguous segment of text containing changes. For each hunk, we count the added and deleted words and take the maximum of the two as the edit count.

For s_2, the average total word count is 703 words, with 241 highlighted words and 188 edited words in s'_2. This means that compared to the total, the reading burden is reduced by 65% and the writing burden by 73%. For s_3, the trend is similar, with the highlighed and edited fractions being even smaller. We conclude that SPECALIGN achieved its objective to carefully limit the burden on the human. We also analyzed the data in more detail and found that, within the highlighted blocks, the average number of deleted words is substantially higher than the number of added words (148 vs. 58). This indicates that the LLM still introduces substantial hallucinated or unnecessary content when rewriting the specification, and developer involvement helps remove such changes. Detailed statistics are provided in Table 9 in the Appendix.

Table 2: Performance of SPECALIGN at Different Levels of Solution Leakage

Leakage category	s_2 (543 instances) Instances (In %)	Resolved @ s_1 (In %)	Resolved @ s_2 (In %)	Change	s_3 (484 instances) Instances (In %)	Resolved @ s_1 (In %)	Resolved @ s_3 (In %)	Change
None	72.5	52.7	69.0	+30.9	94.0	51.4	76.5	+48.7
Minor	2.6	71.4	92.9	+30.0	1.9	77.8	88.9	+14.3
Moderate	13.7	54.1	86.5	+60.0	3.3	56.3	87.5	+55.6
Severe	11.3	47.5	88.5	+86.2	0.8	50.0	75.0	+50.0

4.4 HOW SOLUTION LEAKAGE IMPACTS PERFORMANCE

The simulated developer in our oracle study uses access to the golden code and test patches to address specification gaps identified by SPECALIGN. Although we instruct the LLM to edit only the highlighted parts, access to the golden patches may influence the generated specification and leak solution-specific information both inside and outside the highlighted blocks.

To assess whether our simulated developer (HITL) introduces an unfair advantage for downstream coding agents, we categorize solution leakage into four severity levels: none, minor, moderate, and severe. We evaluate only content that is newly added or modified relative to the initial specification. None represents legitimate specification refinement without solution leakage. Minor loosely resembles the golden patch without revealing the solution. Moderate exposes concrete implementation logic, while severe closely reproduces the golden patch's solution. Pre-existing information, hallucinated content, and information derived solely from the golden test patch are excluded from the leakage assessment. LLMs are useful for software-engineering research annotation tasks, with several LLMs demonstrating reasonable inter-rater agreement (Ahmed et al., 2025a; Wang et al., 2025a). For leakage annotation, we provide each labeling LLM with the initial specification, final specification, and golden code patch. We use three LLMs (Claude-Opus-4.6, GPT-5.4, and Gemini-3.1-Pro) to independently classify the leakage severity. We then use majority voting to determine the final label. If all three models assign different severity levels, we use the label assigned by Claude-Opus-4.6. The leakage annotation prompt is provided in the supplementary material.

Table 2 shows that 72.5% and 94.0% of instances exhibit no leakage for s_2 and s_3, respectively. On the other hand, 11.3% and 0.8% of instances exhibit severe leakage for s_2 and s_3. As expected, performance is higher for instances with severe leakage. Fortunately, performance also improves by 30.9% and 48.7% even for instances with no leakage. These results indicate that the performance improvements achieved by SPECALIGN cannot be attributed solely to solution leakage. SPECALIGN can effectively identify gaps in the specification, and addressing these gaps can improve the coding agent's performance. Measuring inter-rater agreement found Fleiss' κ values of 0.43 and 0.46 for s_2 and s_3, respectively, indicating moderate agreement. Table 11 in the appendix shows detailed inter-rater agreement results.

4.5 EXPECTED CHANGES FROM DEVELOPERS

Table 3: Distribution of Developer Change Categories and Their Impact on Resolution Rate

Description	s_2 (543 instances) Instances (In %)	Resolved @ s_1 (In %)	Resolved @ s_2 (In %)	Change	s_3 (484 intances) Instances (In %)	Resolved @ s_1 (In %)	Resolved @ s_3 (In %)	Change
Clarification / Precision Refinement	30.8	52.1	76.0	+46.0	25.3	50.0	72.1	+44.3
Scope Misspecification	39.4	45.8	71.5	+56.1	18.2	44.3	70.5	+59.0
Mechanism Swap	11.6	60.3	81.0	+34.2	3.7	33.3	88.9	+166.7
Test Specification Fix	16.8	62.6	71.4	+14.0	47.4	55.5	80.8	+45.7
Structural Reorganisation	1.5	87.5	87.5	0.0	5.4	69.2	80.8	+16.7

We showed that incorporating a simulated developer (HITL) can improve the resolution rate. However, what types of changes should we expect from real developers? To provide guidance, we manually examined 100 samples and categorized developer changes into five categories based on the type of specification gap they address. *Clarification/precision refinement* captures small but consequential corrections to ambiguous details, such as identifiers, constants, or ordering constraints. Broader

mismatches between the described and required implementation fall under *scope misspecification*, where developers either remove unnecessary requirements or introduce missing ones. When the scope is correct but the proposed implementation strategy is not, we classify the change as a *mechanism swap*, which includes changes to concurrency primitives, data structures, or error-handling strategies. *Test specification fixes* capture incorrect test-related guidance, such as references to the wrong test file, framework, or helper. Finally, *structural reorganization* covers primarily editorial changes that improve the presentation or organization of the specification without substantially altering its semantics. Fig. 6 to Fig. 10 in the Appendix present examples for each category. We use an annotation approach similar to that described in Section 4.4 to label the remaining samples.

Table 3 shows that the majority of s_2 instances belong to the first two categories. HITL improves the resolution rate across all categories except *structural reorganization*, which accounts for only 1.5% of the instances. We observe a similar improvement for s_3. *Test specification fixes* account for the largest proportion of s_3 instances (47.4%). This is likely because most specification-level details have already been corrected in s_2. At s_3, SPECALIGN primarily focuses on improving alignment, resulting in more test-related refinements. Inter-rater agreement is also higher than in the leakage annotation task, with Fleiss' κ values of 0.58 for s_2 (moderate agreement) and 0.72 for s_3 (substantial agreement). Table 10 in the appendix shows detailed inter-rater agreement results.

4.6 ABLATION STUDY

SPECALIGN depends on the underlying patch and test generators and cannot operate without them. However, we augment the reproduction tests with regression tests, allowing us to evaluate their contribution by removing them. With HITL, we provide our simulated developer with access to both the golden code and test patches. We further investigate how performance changes when only one of these patches is exposed instead of both. To reduce evaluation costs, we conduct the ablation study using Claude-Opus-4.6 on 300 randomly selected instances.

We observe no impact from removing regression tests when HITL is used. This is understandable because HITL helps prevent the LLM from deviating from the intent of the original specification. Without HITL, however, regression tests become important. With test-based selection, removing regression tests reduces performance by 1.2%. With final-patch selection, the performance drop exceeds 11% (detailed results are provided in Table 6 in the Appendix). These results demonstrate the importance of regression tests when the simulated developer is removed from the loop.

Table 4: Impact of Golden Patch Access on Issue Resolution Rates

Selection	Approach	Resolution		Change (In %)
		Count	In %	
Test based	SPECALIGN + HITL	187	62.3	NA
	SPECALIGN + HITL − Golden Test Patch	183	61.0	−2.1
	SPECALIGN + HITL − Golden Code Patch	184	61.3	−1.6
Final patch	SPECALIGN + HITL	230	76.7	NA
	SPECALIGN + HITL − Golden Test Patch	223	74.3	−3.0
	SPECALIGN + HITL − Golden Code Patch	208	69.3	−6.7

Table 4 shows the impact of providing only one golden patch instead of both. As expected, removing the golden code patch has a larger impact than removing the golden test patch. With final-patch selection, performance drops by up to 6.7% when the golden code patch is removed. Nevertheless, the resulting performance remains substantially higher than that achieved without HITL. In particular, achieving a 69.3% resolution rate even without access to the golden code patch suggests that the golden test patch alone provides substantial information that helps the simulated developer address specification gaps. Fig. 5 in the Appendix shows the individual performance of SPECALIGN under different settings.

4.7 PERFORMANCE OF SPECALIGN ON CONTAMINATION-FREE BENCHMARK

SWE-rebench is a SWE-bench-style benchmark that continuously adds new instances to mitigate model memorization and contamination (Badertdinov et al., 2025). We selected 200 instances added after the knowledge cutoff date of Claude-Opus-4.6 (August 2025). We selected instances using the Pytest framework to reduce test-execution related complexity.

Table 5: Performance on Rebench

Approach	Selection	Claude-Opus-4.6 Resolution Count	Resolution In %	Change (In %)
mini-SWE-agent	NA	102	51.0	NA
SPECALIGN	Test based	105	52.5	+2.9
	Final patch	106	53.0	+3.9
SPECALIGN + HITL	Test based	127	63.5	+24.5
	Final patch	176	88.0	+72.5

Table 5 shows that SPECALIGN improves performance by 2.9% to 3.9%, consistent with our observations on SWE-Bench Pro. With HITL, however, we observe larger performance improvements (up to 72.5%) on SWE-rebench, even though the model is unlikely to have encountered these instances during pre-training. Note that, SWE-rebench specifications are relatively shorter than those in SWE-Bench Pro; therefore, even minor modifications can provide substantial additional information to the coding agent.

4.8 COST AND STEP COUNT

Since we iteratively generate code patches, SPECALIGN is expected to be more expensive than running mini-SWE-agent only once. We observe that both the cost and the number of agent steps decrease or remain similar in subsequent iterations when using Claude-Opus-4.6 (see Fig. 4 in Appendix). This is likely because each iteration provides additional hints from test feedback and specification refinement, reducing the search space for the coding agent. However, without HITL, these hints can sometimes be misleading when the refined specification is inaccurate. The average cost of SPECALIGN is $3.04 and $3.29 without and with HITL, respectively. When excluding the cost of test execution and feedback, the average cost decreases to $2.57 and $2.81, respectively.

4.9 ATTRIBUTE-STYLE SPECIFICATION

Considering previous work on attribute-style specifications (REAgent, Kuang et al. (2026)), we implemented an agent to generate a new issue description using an "attribute-style" format. We followed the same attributes of the specification introduced by REAgent, but using mini-SWE-agent to derive the new issue description. However, we do not observe substantial improvement even after incorporating agentic code exploration. Compared with the default specification on 200 instances, SpecAlign with agentic code exploration improves performance by 1.7% without HITL. However, with HITL, performance drops by up to 6.3% (see Table 12 in Appendix). The added attributes can introduce both useful and misleading information. Without HITL, the coding agent can still benefit from some of this information. However, with HITL, where more reliable hints are provided, the benefit of agentic code exploration diminishes and can even negatively affect performance.

5 LIMITATIONS

One limitation of our work is the use of a simulated developer rather than real developers, as recruiting developers with sufficient project-specific expertise is challenging. Nevertheless, our results and leakage analysis provide evidence that addressing specification gaps can improve issue resolution. Our annotation tasks also involve subjectivity; however, reasonable inter-rater agreement across multiple categories provides confidence in the findings. Evaluation on coding benchmarks carries the risk of contamination. We mitigated this risk by also evaluating on a contamination-resistant benchmark. Our results on that benchmark are similar and actually slightly stronger than our main results. Finally, while SPECALIGN effectively identifies specification ambiguities, automatically correcting LLM-induced hallucinations remains challenging. We leave more effective automated specification repair for future work.

6 CONCLUSION

We introduced SPECALIGN, a test-driven approach for minimally refining software specifications by using code and test alignment as a practical signal. SPECALIGN improves coding-agent performance while keeping specification changes small and easy to review. Our HITL study further shows that addressing the specification gaps identified by SPECALIGN can substantially improve issue resolution. Overall, better specification alignment offers a promising path toward generating code that more faithfully reflects human intent.

AI USE STATEMENT

We used AI tools to polish and improve the clarity of the manuscript's writing. We also used AI-based coding agents to assist with code implementation; however, all generated code was reviewed and verified by the authors. AI tools were not used for research idea generation, experimental design, or drawing research conclusions.

REPRODUCIBILITY STATEMENT

The scripts used in this paper are proprietary, and the authors are not permitted to share them without approval from the organization's legal department. However, the prompts used in SPECALIGN, the solution leakage analysis, and the classification of expected developer changes are provided in the supplementary material. We also provide HTML files illustrating the changes made by SPECALIGN and HITL.

REFERENCES

Toufique Ahmed, Premkumar Devanbu, Christoph Treude, and Michael Pradel. Can LLMs replace manual annotation of software engineering artifacts? In *Conference on Mining Software Repositories (MSR)*, pp. 526–538, April 2025a. URL https://doi.org/10.1109/MSR66628.2025.00086.

Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. Otter: Generating tests from issues to validate SWE patches. In *International Conference on Machine Learning (ICML)*, July 2025b. URL https://proceedings.mlr.press/v267/ahmed25b.html.

Toufique Ahmed, Jatin Ganhotra, Avraham Shinnar, and Martin Hirzel. Investigating test overfitting on SWE-bench. In *Conference on the Foundations of Software Engineering: Ideas, Visions, and Reflections track (FSE-IVR)*, pp. 1292–1296, 2026. URL https://doi.org/10.1145/3803437.3805574.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2025. URL https://doi.org/10.52202/085713-0788.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa R Kundurthy, Sean M. Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-bench pro: Can AI agents solve long-horizon software engineering tasks? In *International Conference on Machine Learning (ICML)*, 2026. URL https://openreview.net/forum?id=uEVTdoAbnK.

Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, Yun Lin, Yingfei Xiong, Chao Peng, and Xia Liu. Trae agent: An LLM-based agent for software engineering with test-time scaling, July 2025. URL https://arxiv.org/abs/2507.23370.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, May 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Lara Khatib, Noble Saji Mathews, and Meiyappan Nagappan. AssertFlip: Reproducing bugs via inversion of LLM-generated passing tests. In *International Conference on Software Engineering (ICSE)*, April 2026a. URL https://doi.org/10.1145/3744916.3773127.

Lara Khatib, Noble Saji Mathews, Meiyappan Nagappan, Pengyu Nie, and Thomas Zimmermann. What makes a good bug report for an AI agent?, July 2026b. URL https://arxiv.org/abs/2607.07593.

Shiqi Kuang, Zhao Tian, Kaiwei Lin, Chaofan Tao, Shaowei Wang, Haoli Bai, Lifeng Shang, and Junjie Chen. REAgent: Requirement-driven LLM agents for software issue resolution, April 2026. URL https://arxiv.org/abs/2604.06861.

Shuvendu K. Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madanlal Musuvathi, Piali Choudhury, Curtis von Veh, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. Interactive code generation via test-driven user-intent formalization, August 2023. URL https://arxiv.org/abs/2208.05950.

Qingyuan Liu, Mo Zou, Hengbin Zhang, Dong Du, Yubin Xia, and Haibo Chen. Sharpen the spec, cut the code: A case for generative file system with SYSSPEC. In *USENIX Conference on File and Storage Technologies (FAST)*, pp. 291–311, February 2026. URL https://www.usenix.org/conference/fast26/presentation/liu-qingyuan.

Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. SpecRover: Code intent extraction via LLMs. In *International Conference on Software Engineering (ICSE)*, May 2025. URL https://doi.org/10.1109/ICSE55347.2025.00080.

Manan Suri, Xiangci Li, Mehdi Shojaie, Songyang Han, Chao-Chun Hsu, Shweta Garg, Aniket Anand Deshmukh, and Varun Kumar. CodeScout: Contextual problem statement enhancement for software agents. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 40902–40931, July 2026. URL https://doi.org/10.18653/v1/2026.findings-acl.2032.

Wannita Takerngsaksiri, Jirat Pasuksmit, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Ruixiong Zhang, Fan Jiang, Jing Li, Evan Cook, Kun Chen, and Ming Wu. Human-in-the-loop software development agents. In *International Conference on Software Engineering: Software Engineering in Practice track (ICSE-SEIP)*, pp. 342–352, April 2025. URL https://doi.org/10.1109/ICSE-SEIP66354.2025.00036.

Junyi Wang, Jialun Cao, and Zhongxin Liu. iCoRe: An iterative correlation-aware retriever for bug reproduction test generation. In *Symposium on the Foundations of Software Engineering (FSE)*, July 2026. URL https://doi.org/10.1145/3808193.

Ruiqi Wang, Jiyu Guo, Cuiyun Gao, Guodong Fan, Chun Yong Chong, and Xin Xia. Can LLMs replace human evaluators? an empirical study of LLM-as-a-judge in software engineering. In *International Symposium on Software Testing and Analysis (ISSTA)*, June 2025a. URL https://doi.org/10.1145/3728963.

Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. AEGIS: An agent-based framework for general bug reproduction from issue descriptions, November 2024. URL https://arxiv.org/abs/2411.18015.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi 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. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, pp. 65882–65919, April 2025b. URL https://openreview.net/forum?id=OJd3ayDDoF.

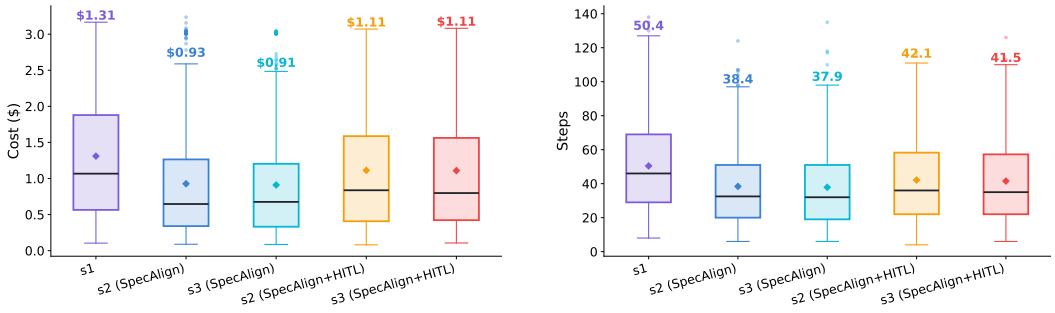
John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-Agent: Agent-computer interfaces enable automated software engineering. In *Conference on Neural Information Processing Systems (NeurIPS)*, pp. 50528–50652, December 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *International Symposium on Software Testing and Analysis (ISSTA)*, pp. 1592–1604, September 2024. URL https://doi.org/10.1145/3650212.3680384.

A APPENDIX

A.1 AGENT COST AND STEP COUNT

The cost and step-count box plots compare the computational effort of the coding agent across different iterations (see Fig. 4). Overall, they show how iterative specification refinement affects both the monetary cost and the number of agent steps required to generate a patch. These results help quantify the additional overhead introduced by SPECALIGN.



(a) Cost (b) Steps

Figure 4: Cost and Agent Step Count Across Different Specification Stages

A.2 IMPACT OF REGRESSION TESTS ON SPECALIGN

Table 6 studies the impact of regression tests on SPECALIGN. Regression tests help ensure that generated patches preserve existing functionality while addressing the target issue. The results show how including regression tests affects specification refinement and downstream issue resolution.

Table 6: Impact of Regression Test on SPECALIGN

Selection	Approach	Resolution Count	Resolution In %	Change In %
Test-based	SPECALIGN	172	57.3	NA
	SPECALIGN- Regression	170	56.7	-1.2
Final-patch	SPECALIGN	170	56.7	NA
	SPECALIGN- Regression	151	50.3	-11.2
Test-based	SPECALIGN + HITL	187	62.3	NA
	SPECALIGN + HITL- Regression	187	62.3	0.0
Final-patch	SPECALIGN + HITL	230	76.7	NA
	SPECALIGN + HITL- Regression	230	76.7	0.0

A.3 COMPARISON OF PATCH SELECTION STRATEGIES

Table 7 and Table 8 compare the two patch-selection strategies used in SPECALIGN. Test-based selection chooses a patch based on the generated test results, while final-patch selection uses the patch from the latest specification refinement. These results show how the choice of patch-selection strategy affects the overall issue resolution performance.

A.4 ANALYSIS OF SPECIFICATION EDITS ACROSS ITERATIONS

Table 9 provides descriptive statistics of the specification edits between s_2 and s_3. We compare changes in word count, number of edit hunks, additions, and deletions. The results characterize the size and extent of specification refinements across iterations.

A.5 INTER-RATER AGREEMENT ANALYSIS

Table 10 and Table 11 report inter-rater agreement for two manual annotation tasks: ground-truth information leakage and the types of specification changes made during refinement. The first evalu-

Table 7: Performance of SPECALIGN (with and without HITL) using test-based selection

Language	Approach	Claude-Opus-4.6			GPT-5.4		
		Resolution		Change	Resolution		Change
		Count	In %	(In %)	Count	In %	(In %)
Python (266)	mini-SWE-agent	167	62.8	NA	139	52.3	NA
	SPECALIGN	172	64.7	+3.0	136	51.1	-2.2
	SPECALIGN + HITL	192	72.2	+15.0	142	53.4	+2.2
Go (280)	mini-SWE-agent	144	51.4	NA	109	38.9	NA
	SPECALIGN	148	52.9	+2.8	120	42.9	+10.1
	SPECALIGN + HITL	150	53.6	+4.2	117	41.8	+7.3
JS+TS (185)	mini-SWE-agent	114	61.1	NA	85	45.9	NA
	SPECALIGN	115	62.2	+0.9	89	48.1	+4.7
	SPECALIGN + HITL	127	68.6	+11.4	87	47.0	+2.4
Overall (731)	mini-SWE-agent	425	58.1	NA	333	45.6	NA
	SPECALIGN	435	59.5	+2.4	345	47.2	+3.6
	SPECALIGN + HITL	469	64.2	+10.4	346	47.3	+3.9

Table 8: Performance of SPECALIGN (with and without HITL) using final-patch selection

Language	Approach	Claude-Opus-4.6			GPT-5.4		
		Resolution		Change	Resolution		Change
		Count	In %	(In %)	Count	In %	(In %)
Python (266)	mini-SWE-agent	167	62.8	NA	139	52.3	NA
	SPECALIGN	174	65.4	+4.2	133	50.0	-4.3
	SPECALIGN + HITL	213	80.1	+27.5	154	57.9	+10.8
Go (280)	mini-SWE-agent	144	51.4	NA	109	38.9	NA
	SPECALIGN	148	52.9	+2.8	107	38.2	-1.8
	SPECALIGN + HITL	207	73.9	+43.8	121	43.2	+11.0
JS+TS (185)	mini-SWE-agent	114	61.1	NA	85	45.9	NA
	SPECALIGN	110	59.5	-3.5	94	50.8	+10.6
	SPECALIGN + HITL	151	81.6	+32.5	104	56.2	+22.4
Overall (731)	mini-SWE-agent	425	58.1	NA	333	45.6	NA
	SPECALIGN	432	59.1	+1.6	334	45.7	+0.3
	SPECALIGN + HITL	571	78.1	+34.4	379	51.8	+13.8

ates how consistently annotators identify whether HITL refinements introduce information from the ground truth, while the second evaluates agreement when categorizing the types of changes made to the specification.

A.6 IMPACT OF ATTRIBUTE-STYLE SPECIFICATION

Table 12 presents the results of the attribute-based analysis of specification refinements.

A.7 PERFORMANCE OF SPECALIGN FOR s_2 AND s_3 WITH AND WITHOUT HITL

Fig. 5 presents the individual performance of SPECALIGN for s_2 and s_3 under different settings. It also presents the resolution rates from the ablation study.

A.8 EXAMPLES FOR EXPECTED CHANGES FROM DEVELOPERS.

Fig. 6 to Fig. 10 show examples of different classes of developer changes.

Table 9: Descriptive statistics and edit-level diff metrics (word counts, hunk counts, additions, and deletions) comparing s_2 and s_3 problem descriptions.

Metric	s_2						s_3					
	n	Mean	Median	Min	Max	± 95% CI	n	Mean	Median	Min	Max	± 95% CI
Total Word Count	543	703.7	649	175	1906	± 21.3	484	960.9	921.5	191	2413	± 29.7
Highlighted Block Count	543	2.5	2	1	15	± 0.2	484	2.2	2.0	1	14	± 0.1
Highlighted Word Count	543	241.4	216	27	809	± 10.9	484	218.4	196.5	13	1701	± 12.7
Hunk count	543	21.6	15	0	133	± 1.9	484	16.5	10.0	0	156	± 1.8
Added Word Count	543	58.0	33	0	506	± 6.1	484	42.1	22.5	0	615	± 5.4
Deleted Word Count	543	148.7	126	0	760	± 10.3	484	111.1	85	0	1,701	± 11.3
Updated Word Count	543	188.0	166	0	760	± 11.0	484	138.9	110.5	0	1701	± 12.3

Table 10: Inter-Rater Agreement for Solution Leakage Annotation

Pair	s_2			s_3		
	% Agreement	Cohen's Kappa	Interpretation	% Agreement	Cohen's Kappa	Interpretation
Gemini vs GPT	67.6	0.54	Moderate	80.3	0.71	Substantial
Gemini vs Claude	75.9	0.66	Moderate	82.4	0.74	Substantial
GPT vs Claude	67.4	0.54	Substantial	80.3	0.71	Substantial
All	NA	0.58*	Moderate	NA	0.72*	Substantial

*Fleiss' Kappa is used because of three raters

Table 11: Inter-Rater Agreement for Solution Leakage Annotation

Pair	s_2			s_3		
	% Agreement	Cohen's Kappa	Interpretation	% Agreement	Cohen's Kappa	Interpretation
Gemini vs GPT	77.7	0.43	Moderate	95.2	0.54	Moderate
Gemini vs Claude	74.4	0.53	Moderate	91.7	0.47	Moderate
GPT vs Claude	66.4	0.34	Fair	90.7	0.40	Moderate
All	NA	0.43*	Moderate	NA	0.46*	Moderate

*Fleiss' Kappa is used because of three raters

Table 12: Impact of Attribute-style Specification Writing

Selection	Approach	Default	In %	Attribute	In %	Change In %
Test based	SPECALIGN	120	60.0	122	61.0	+1.7
	SPECALIGN + HITL	127	63.5	119	59.5	-6.3
Final patch	SPECALIGN	116	58.0	118	59.0	+1.7
	SPECALIGN + HITL	151	75.5	145	72.5	-4.0

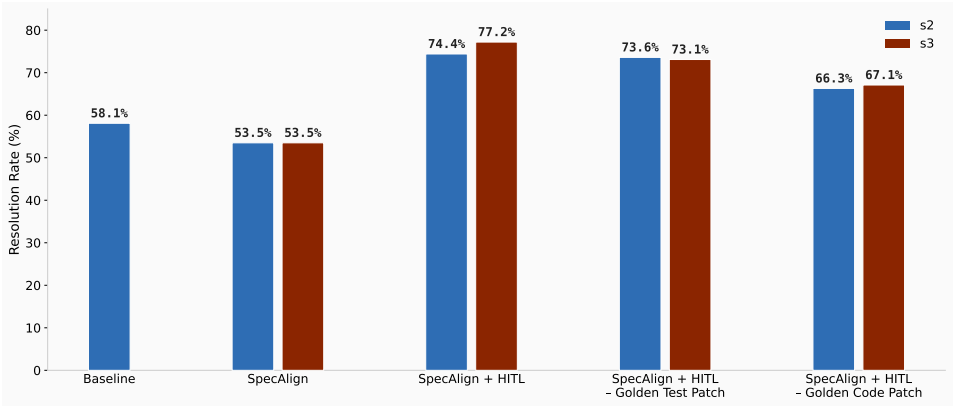


Figure 5: Individual Performance of SPECALIGN under Different Settings

NodeBB__NodeBB-da0211b1

INIT said: use hostname after @ in the WebFinger URI

✓ FINAL: must use host (which includes the port number) — validation uses host, not hostname

↳ *hostname strips the port; host preserves it. URI validation fails for non-80/443 ports.*

ansible__ansible-395e5e20

INIT said: version set to '2.12'

✓ FINAL: must be '2.14' — this is a literal constant in the golden patch

↳ *Version string is included in the emitted deprecation warning; tests that parse the warning text will fail.*

flipt-io__flipt-21a935a

INIT said: JSON tag: "grpcLevel,omitempty"

✓ FINAL: must be "grpc_level,omitempty" — snake_case convention used throughout the config system

↳ *camelCase tag means the YAML field 'grpc_level' is never read; the value stays at zero silently.*

element-hq__element-web-33e8edb3

INIT said: always apply toLowerCase() to both sides of the key comparison

✓ FINAL: only apply toLowerCase() when ev.shiftKey is true; use exact comparison otherwise

↳ *Always case-folding breaks combos like Ctrl+K that must match lowercase 'k' exactly.*

Figure 6: Examples of Clarification/Precision Refinement

ansible__ansible-189fcb37

INIT said: modify WapiModule.run() with special obj_filter handling for fixed address types

✓ FINAL: run() does NOT need modification — only get_object_ref() needs the new elif branch

↳ *Broader method scope (run) vs narrower (get_object_ref) — changing the wrong method breaks existing tests.*

NodeBB__NodeBB-0f788b8e

INIT said: deleteAll sets numThumbs to 0 via topics.setTopicField

✓ FINAL: deleteAll must delegate to Thumbs.delete — all numThumbs decrement happens inside that method

↳ *The extra setTopicField call would double-update the counter and put it out of sync.*

element-hq__element-web-1077729a

INIT said: add special null-handling early-return guard in updateRoomsAndIndex

✓ FINAL: existing ?? operator handles null correctly — guard not needed; requirement removed

↳ *The guard would short-circuit on null when the ?? operator already falls back gracefully.*

flipt-io__flipt-756f00f

INIT said: append Warnings []string field to the existing Config struct

✓ FINAL: Warnings must be on a separate Result struct — adding to Config is over-scope

↳ *Putting Warnings on Config contaminates the config type with operational state; callers that serialize Config would also serialize warnings.*

Figure 7: Examples of Scope Misspecification

ansible__ansible-b5e0293

INIT said: character-level regex _CLIXML_HEX_RE on decoded strings, with explicit surrogate pair handling

✓ FINAL: UTF-16-BE byte-level regex _STRING_DESERIAL_FIND operating on encoded bytes — surrogates handled naturally

↳ *Operating on bytes before decoding sidesteps the surrogate pair problem entirely; explicit surrogate logic is error-prone.*

NodeBB__NodeBB-3c85b944

INIT said: sequential admin check, then isMemberOfAny if admin fails

✓ FINAL: Promise.all([adminCheck, isMemberOfAny]) with OR condition — parallel, not sequential

↳ *Sequential short-circuits on admin=true but still awaits one promise; parallel is both correct and faster.*

navidrome__navidrome-66b74

INIT said: truncate key if longer than 32 bytes, zero-pad if shorter

✓ FINAL: crypto/sha256.Sum256() to hash the key string into exactly 32 bytes — idiomatic Go key derivation

↳ *Truncation/padding changes the effective key for existing encrypted data; sha256 produces a deterministic 32-byte key that matches what was used to encrypt.*

gravitational__teleport-0ecf31

INIT said: ReadPassword handles context cancellation and terminal restoration internally

✓ FINAL: terminal restoration handled externally by fireCleanRead and Close — ReadPassword itself is unchanged

↳ *Putting cleanup inside ReadPassword mixes read and cleanup responsibilities; the golden separates them into lifecycle methods.*

Figure 8: Examples of Mechanism Swap

ansible__ansible-83fb24b9

INIT said: create a new separate test file for the multiport tests

✓ FINAL: must be added to existing TestIptables class in test/units/modules/test_iptables.py — that class's setUp mocks get_bin_path

↳ *A new file has no setUp infrastructure; get_bin_path is not mocked, causing the test to fail with a FileNotFoundError.*

INIT said: create a new standalone test file

✓ FINAL: must be added to test/posts.js inside the voting describe block, using shared voterUid, postData, cid variables

↳ *A standalone file with a single user would cause a self-vote scenario that the privilege check does not cover.*

flipt-io__flipt-3b2c25ee

INIT said: separate test file for the ofrep evaluation test cases

✓ FINAL: must be part of internal/server/ofrep/evaluation_test.go using package ofrep — needs access to internal constructor

↳ *Package-level access is needed to call the internal store mock constructor.*

protonmail__webclients-01ea52

INIT said: test file with .test.js extension

✓ FINAL: must be .test.ts (TypeScript) — build chain rejects .js files in this repository

↳ *The TypeScript compiler enforces strict file extension conventions; .js files are excluded from the TS compilation step.*

Figure 9: Examples of Test Specification Fix

ansible__ansible-5e88cd9

▶ **INIT said:** **<new>** tags **wrapping** module **attribute** **assignment;** **module** **set as string path**

✓ **FINAL:** tags **removed; note added: 'module must be set to the actual imported** module **object, not a string'**

↳ *The tag removal is structural; the clarification (object vs string) is technically Cat 1, but the change is dominated by the tag removal.*

element-hq__element-web-8f3c8b35

▶ **INIT said:** **'regenerate** the snapshot **file' (vague)**

✓ **FINAL:** **'add aria-label="test" to** the **outer div in the ExtraTile renders 1** snapshot **in ExtraTile-test.tsx.snap'**

↳ *Prose generalisation → precise code-level instruction. The meaning is the same but the actionability is much higher.*

internetarchive__openlibrary-4b7ea2

▶ **INIT said:** 'add **logic to copy** remote_ids **field** to the **result dict' (prose)**

✓ **FINAL:** 'add remote_ids to the **existing for f in "name", "title", ... tuple' — points to the exact code line**

↳ *Same semantic intent (copy the field) but the prose was ambiguous enough to lead to a different implementation approach.*

gravitational__teleport-e6681a

▶ **INIT said:** **AuditBackoffTimeout and AsyncBufferSize as const** declarations

✓ **FINAL:** **must be var** declarations **in the existing var block of lib/defaults/defaults.go**

↳ *const vs var is a precision fix but is a single word change in a section label — structurally minor.*

Figure 10: Examples of Structural Reorganisation