NoCode-bench: Evaluating Natural Language-Driven Software Feature Addition

Anonymous ACL submission

Abstract

LLMs have demonstrated remarkable capabilities in supporting software developers, e.g., by automating code generation and code editing. In contrast, their effectiveness and limitations in enabling software users to incrementally improve a piece of software are currently underexplored. A promising paradigm toward this end is *natural language-driven feature addition*, which allows users to specify and modify software functionality purely through natural language (NL) descriptions, sometimes also called "no-code development". This paper introduces NoCode-bench, a benchmark designed to evaluate LLMs on real-world NL-driven software feature addition tasks. NoCode-bench consists of 634 tasks across 10 popular projects, each of which pairs a user-oriented documentation change and the corresponding code implementation that can be validated against developer-written test cases. To facilitate lightweight and reliable evaluation, we further curate a human-validated subset named NoCode-bench Verified. It covers 114 high-quality tasks across projects, where the task clarity and evaluation validity are manually verified. We use NoCode-bench to assess a range of state-of-the-art LLMs. Experimental results show that despite significant token consumption, the best task success rate remains as low as 37.72% when using the OpenHands scaffold combined with Qwen3-Coder-480B. Our analysis reveals that LLMs face key challenges in performing cross-file edits, understanding existing code modules, and accurately calling tools.

1 INTRODUCTION

Large language models (LLMs) have demonstrated remarkable capabilities in supporting software developers across a wide range of programming tasks (Xu et al., 2022; Hou et al., 2024), including code generation, test generation, and program repair. These advances have led to the widespread adoption of LLM-powered software engineering tools (Kumar et al., 2025; Pudari and Ernst, 2023), where models are typically prompted with rich code context and explicit instructions to generate or modify source code (Xia et al., 2025; Wang et al., 2025). As a result, LLMs are increasingly viewed as versatile programming assistants that can automate a wide range of developer-oriented activities.

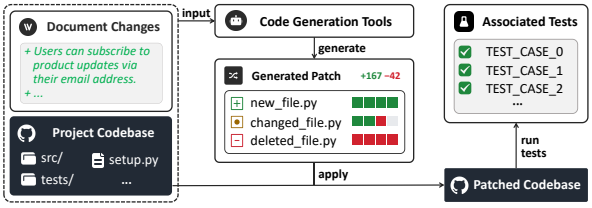

Figure 1: Overview of NL-driven feature addition tasks.

In contrast, the effectiveness and limitations of LLMs in supporting software users, especially in incrementally improving software, are currently underexplored. Software users typically hope to add or modify specific features in existing software to meet their needs, yet usually lack familiarity with the underlying codebase or development tools (Sahay et al., 2020; Rao et al., 2024). A promising paradigm toward such user-oriented scenarios is natural language-driven feature addition. This paradigm allows users to specify new software functionality or behavioral changes purely through natural language (NL) descriptions with the underlying code automatically generated by the no-code system (Hirzel, 2023), sometimes also called "no-code development".

To understand and improve the performance of LLMs in NL-driven feature addition, there is a need for high-quality benchmarks. Although several benchmarks have been proposed to evaluate LLMs on software engineering problems (Jiang et al., 2025), they exhibit a substantial gap with respect to no-code scenarios. Specifically, they predominantly use issue reports as task inputs, which are designed for coordination among developers (GitHub,

Inc., 2025a) and are typically authored by active code contributors (Bissyandé et al., 2013). Prior studies (Chaparro et al., 2019; Soltani et al., 2020) also show that high-quality issue and bug reports often include implementation-level technical details such as reproduction steps, stack traces, and test cases, implicitly assuming familiarity with system internals. Such developer-centric inputs differ fundamentally from those in no-code development, where requirements are expressed in natural language without assuming programming expertise. Moreover, existing benchmarks largely focus on bug fixing or issue resolution, while paying limited attention to feature addition, despite feature addition being a central activity and accounting for around 60% of software maintenance efforts (Rahman, 2019; Glass, 2001).

To fill this gap, we present NoCode-bench, a new benchmark designed to evaluate LLMs' capability in NL-driven feature addition. NoCode-Bench identifies feature addition tasks through release notes and constructs natural language specifications from *changes in user-facing software documentation*, which serves as an authoritative and comprehensive NL specification of software functionality (Aghajani et al., 2019). It consists of 634 carefully collected tasks derived from high-quality open-source projects hosted on GitHub. As illustrated in Figure 1, each instance takes documentation changes as input and expects the model to generate corresponding code changes. The implementation is validated using developer-written test cases. In addition, to provide a lightweight and reliable evaluation option, we manually validate a subset called NoCode-bench Verified. This subset consists of 114 instances with their task clarity and evaluation accuracy manually verified, enabling reliable evaluation under limited resources.

We leverage both NoCode-bench Verified and NoCode-bench Full to systematically evaluate a range of state-of-the-art LLMs with the Agentless and OpenHands frameworks. Evaluation results show that, despite incurring high token consumption, the best success rate of existing models is only 37.72%, achieved when using Qwen3-Coder-480B with the OpenHands scaffold. In comparison, when evaluated on SWE-bench, a developer-centric benchmark collected from similar projects, the same model and scaffold achieve a success rate of 69.60%. This stark performance gap highlights the substantial challenges of user-centric scenarios and no-code development. Our analysis further reveals that LLMs face key challenges in performing cross-file edits, understanding existing code modules, and accurately calling tools when completing NL-driven feature addition tasks. Our code and data are available at `https://anonymous.4open.science/r/NoCode-bench-DFBC`.

In summary, our main contributions are:

- We introduce NoCode-bench, a novel user-centric benchmark that evaluates NL-driven software feature addition, paving the way for no-code software development.
- We present a systematic five-phase construction pipeline to ensure the quality and fidelity of the benchmark, and further provide a manually validated subset for lightweight and reliable evaluation under limited resources.
- We comprehensively evaluate multiple state-of-the-art LLMs on NoCode-bench, analyze their performance from both quantitative and qualitative perspectives, and identify key factors that affect model performance.

2 NoCode-bench

NoCode-bench focuses on evaluating LLMs in NL-driven feature-addition scenarios. It comprises 634 tasks collected from 10 popular open-source projects. Each task corresponds to a real-world feature addition explicitly documented in the project's official release notes. Each feature addition includes a coordinated change to the documentation, the code change, and the test suite. The goal of each task is to produce a patch that implements the feature specified by the documentation change and passes the relevant tests. The input includes the documentation changes and the complete codebase, as illustrated in Figure 1. The task design aligns with documentation-first development practices, such as README-driven development (Preston-Werner, 2010), OpenAPI-based workflows (OpenAPI, 2025), and Spec-driven development (GitHub, Inc., 2025b), where documentation changes precede and guide code modifications.

2.1 Benchmark Construction

We focus on high-quality, real-world software development data to construct the NoCode-bench through five phases (Figure 2):

Phase 1: Project Selection. The project selection begins with the 12 projects used in SWE-bench (Jimenez et al., 2024), which are actively

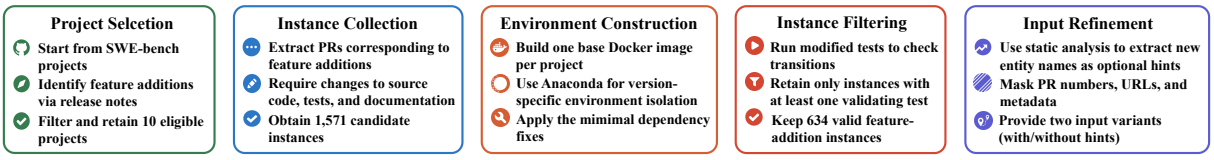

Figure 2: The workflow of building NoCode-bench.

maintained and well-documented open-source projects. Unlike SWE-bench, which constructs instances by linking GitHub issues to corresponding pull requests (PRs), we identify real-world feature addition tasks from release notes, which are written or confirmed by project developers and typically include clearly labeled feature additions, making them a reliable source of accurate task annotations. To ensure that the benchmark is reliable, representative, and suitable for NL-driven feature addition, we apply additional criteria when selecting projects. Eligible projects must: (1) provide release notes that signal sustained maintenance (Bi et al., 2020; Yu, 2009); (2) label each change item with its type (e.g., feature, enhancement), which supports accurate identification of feature-addition instances; and (3) link every change item to at least one GitHub PR, allowing us to trace documented features back to their corresponding code modifications. Following this approach, we evaluate all 12 projects from SWE-bench and find that 10 of them meet all the above conditions. The `flask` (Pallets, 2025) and `sympy` (Team, 2025) projects lack feature-labeled annotations and are thus excluded.

Phase 2: Instance Collection. For each project selected in Phase 1, we retrieve all PRs corresponding to the feature additions identified from release notes, resulting in a total of 3,263 PRs. To ensure that every instance reflects a genuine, verifiable feature update, we retain only PRs that simultaneously modify (1) source code, (2) test files, and (3) user-facing documentation. Unlike SWE-bench, which considers only source code and test modifications, we additionally require documentation changes, which serve as the NL specification provided as input to our task. Compared to issue reports, it better aligns with the no-code setting, as they describe the intended feature behavior without exposing implementation details. For each retained PR, we record the base commit, the documentation diff (i.e., the input of the NL-driven feature addition task), and the feature and test patches (i.e., the evaluation oracle). Applying these criteria yields 1,571 candidate instances, which are then processed.

Phase 3: Environment Construction. To evaluate feature implementations reliably, each instance must be executed in a reproducible environment. Unlike SWE-bench, which builds one Docker image per instance, we adopt a more scalable strategy (Yang et al., 2025c): one base Docker image per project, with version-specific isolation handled through lightweight environment management, i.e., Anaconda (Rolon-Mérette et al., 2016). This design leverages the fact that instances within a project usually share similar dependencies, reducing redundancy while preserving reproducibility. For each project, we reconstruct the environment corresponding to the feature's base commit and install its declared dependencies. When discrepancies arise, such as unpinned or outdated library versions, we apply minimal manual fixes to restore executability. Because such issues often recur across instances of the same project, a small number of fixes typically generalize broadly. This phase ensures that all tasks run in clean and consistent containerized environments, providing a reliable foundation for automated test-based evaluation.

Phase 4: Instance Filtering. Then, we verify that each collected PR indeed corresponds to a valid feature addition. For every instance, we execute the test files modified or introduced in the PR both before and after applying the feature patch. Tests that remain passed (PASSED→PASSED) ensure regression safety, while tests that transition from failing to passing (FAILED→PASSED) serve as evidence that the new feature has been implemented. We denote these two categories as F→P tests and P→P tests, respectively. An instance is kept only if it contains at least one such validation test, providing a clear behavioral signal of feature emergence. Unlike SWE-bench, we do not discard instances whose pre-patch executions raise errors (e.g., *ImportError* or *AttributeError*), as such failures naturally arise when a feature introduces previously undefined modules, classes, or functions. Applying this filtering yields 634 validated feature addition instances used in NoCode-bench.

Phase 5: Input Refinement. During benchmark

construction, we observed that some feature additions introduce new program entities (e.g., files, classes, or methods) that are referenced in tests and implementation patches but not mentioned in the documentation change. As the evaluation is test-driven, mismatched or missing entity names can cause models to fail tests even when the generated feature is correct. Simply discarding such cases, as done in prior work (Jimenez et al., 2024), would remove legitimate feature additions and thus reduce benchmark coverage. To preserve these instances while avoiding superficial evaluation failures, we extract the names of newly introduced, test-referenced code entities, such as classes and methods, through lightweight static analysis. These names can be appended as optional hints to the model input to reduce false negatives, or omitted for a stricter evaluation setting. We release both benchmark variants to support different evaluation preferences. Additionally, we automatically mask PR numbers, URLs, and other metadata in documentation diffs to prevent models from leveraging memorized or externally fetchable information.

Together with previous phases, this yields the final set of 634 NL-driven feature addition tasks used in NoCode-bench. A final breakdown of these task instances across projects is presented in Table 1. Technical details about NoCode-bench's construction pipeline are discussed in Appendix A.1.

2.2 NoCode-bench Verified

To complement the full benchmark of 634 instances (NoCode-bench Full) and support lightweight, reliable evaluation, we construct a manually validated subset, NoCode-bench Verified. This subset is intended for fine-grained debugging, controlled studies of model behavior, and scenarios with limited computational budgets.

Construction Process. We aim to preserve diversity across projects and target a subset of roughly 100 high-quality instances. We sample candidate tasks from all projects and perform double-blind manual verification following the guidelines of SWE-bench Verified (Chowdhury et al., 2024). Five experienced developers independently assess each sampled instance.

Annotation Criteria The annotation adopts two criteria: *task clarity* and *evaluation accuracy*. Both criteria use a coarse 0-3 scale, with lower scores indicating higher quality. The task clarity measures whether a competent engineer, with access to the codebase, can reasonably infer the expected implementation from the updated documentation. The evaluation accuracy reflects whether the test patch provides appropriate and targeted functional coverage of the described feature. Annotators may additionally flag instances exhibiting other major issues.

Table 1: Instance counts of each project in NoCode-Bench Full and NoCode-Bench Verified.

Project	Owner	Full	Verified
seaborn	mwaskom	28	13
scikit-learn	scikit-learn	234	16
xarray	pydata	87	17
astropy	astropy	164	19
django	django	24	8
pylint	pylint-dev	16	6
sphinx	sphinx-doc	51	17
pytest	pytest-dev	4	1
requests	psf	2	1
matplotlib	matplotlib	24	16
Total	–	**634**	**114**

Annotation Results We discard instances whose task clarity or evaluation accuracy receives a score of 2 or higher, as well as any instance flagged for major issues. This process removes 124 candidates and yields 114 verified instances. Inter-annotator agreement, measured by Cohen's kappa, is 0.42, which is comparable to 0.39 in SWE-bench Verified and indicative of moderate but acceptable reliability.

NoCode-bench Verified thus offers a curated, dependable subset for controlled LLM evaluation while remaining representative of documentation-driven feature addition scenarios. The complete annotation process and results are documented in Appendix A.2.

3 Benchmark Characteristics

To better understand the unique characteristics and challenges posed by NoCode-bench, we analyze the distribution of instances in NoCode-bench and SWE-bench across dimensions related to input complexity, localization difficulty, and editing difficulty, as shown in Figure 3. For better visualization, we remove the top 1% of the data in Figure 3, but we keep them during the following analysis.

Input complexity. Unlike SWE-bench, which uses issue descriptions as input, NoCode-bench takes document changes as input and requires the LLM to understand the semantic differences before and after the change. Long document changes pose extra challenges for the LLM to comprehend the

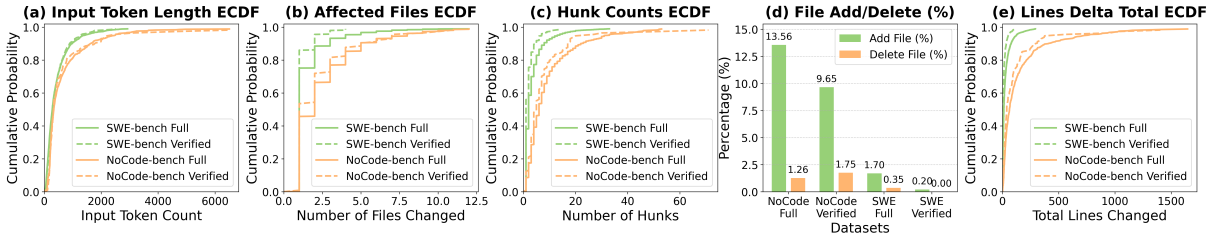

Figure 3: Comparison of golden patch distributions between NoCode-bench and SWE-bench along dimensions related to localization and editing difficulty. ECDF stands for "empirical cumulative distribution function".

task and generate appropriate outputs. As shown in Figure 3 (a), the average length of issue descriptions in SWE-bench Full and Verified is 480.37 and 447.72, respectively, while the average length of document changes in NoCode-bench Full and Verified is 739.06 and 820.65, which is nearly twice as long as that in SWE-bench. This indicates that tasks in NoCode-bench require LLMs to capture information from longer and more complex task inputs, posing greater challenges to their abilities in long-text comprehension, identifying key information, and attending to fine-grained details.

Localization Difficulty. Accurate localization in the codebase is a necessary condition for generating the correct patches. Localization is particularly challenging when multiple files are involved or when the changes are scattered across many code regions. As shown in Figure 3 (b,c), on average, golden patches in NoCode-bench Full involve edits to 2.65 files and 10.32 hunks, whereas in SWE-bench Full, they involve only 1.66 files and 4.12 hunks. In the Verified subsets, the number of files involved is 2.39 for NoCode-bench and 1.24 for SWE-bench, with corresponding hunk counts of 8.96 and 2.43. Moreover, Figure 3 (d) reveals that NoCode-bench includes a much higher proportion of patches that involve adding or deleting files. 13.56% of instances in NoCode-bench Full require new files, while the percentage is only 1.70% in SWE-bench Full. These results indicate that our benchmark requires LLMs to locate multiple code blocks spread across different files within large codebases, which goes far beyond the demands of issue-solving tasks in SWE-bench, leading to localization complexity.

Editing Difficulty. Once the correct edit location is identified, the correct code segments must be generated to realize the intended functionality. Larger edits typically increase the chance of introducing syntactic or semantic errors. We measure the total number of lines changed in golden patches, i.e., the ground truth edits excluding changes to tests and documentation files, as shown in Figure 3 (e). In NoCode-bench Full, golden patches involve an average of 179.12 lines of edits, and 179.22 lines in NoCode-bench Verified. In contrast, golden patches in SWE-bench Full average 37.71 lines of edits, and only 14.32 lines in SWE-bench Verified. Furthermore, we find that SWE-bench patches are small, with nearly all patches under 200 lines. In contrast, NoCode-bench includes a substantial portion of larger edits, with nearly 20% of patches exceeding 200 lines. These larger modifications may require stronger code generation capabilities and greater coherence across multi-line edits.

4 Experimental Setup

Model Selection. Completing a task in NoCode-bench typically requires a deep understanding of the target project, demanding strong reasoning ability and the capacity to handle long contexts. Considering these requirements, we selected 8 state-of-the-art LLMs for evaluation on NoCode-bench Verified. Four of them are open-source models, i.e., DeepSeek-v3-0324 (Liu et al., 2024), DeepSeek-R1-0528 (Guo et al., 2025a), Qwen3-235B-A22B (Yang et al., 2025a), and Qwne3-Coder-480B-A35B, where DeepSeek-R1 is a reasoning model and Qwen3-235B-A22B is used in thinking mode. The other four are closed-source models, i.e., Claude-4-Sonnet (Anthropic, 2024), GPT-4o-2024-11-20 (OpenAI, 2024), Gemini-2.5-Pro-Exp-03-25 (Comanici et al., 2025), and GPT-5-mini (OpenAI, 2025), where Gemini-2.5-Pro is a reasoning model and GPT-5-mini is used in medium thinking. For brevity, version numbers will be omitted when referencing model names throughout the paper. NoCode-bench Full is significantly larger than NoCode-bench Verified, with 5.6 times the number of instances. Due to the high cost of Claude-4-Sonnet, Qwen3-Coder, GPT-5-mini, and Gemini-2.5-Pro (total estimated at

Table 2: Performance comparison across different LLMs on NoCode-bench Verified.

Model	Agentless							OpenHands						
	Success%	Applied%	RT%	FV-Micro	FV-Macro	File%	#Token	Success%	Applied%	RT%	FV-Micro	FV-Macro	File%	#Token
Open-Source Model														
Qwen3-235B	13.16%	**100.00%**	76.32%	8.75%	22.39%	42.98%	0.12M	7.89%	64.91%	47.37%	1.96%	14.03%	45.61%	0.24M
DeepSeek-v3	21.05%	98.25%	78.95%	7.96%	32.80%	57.14%	0.30M	11.40%	65.79%	49.12%	1.68%	18.29%	56.14%	0.59M
DeepSeek-R1	25.44%	93.86%	73.68%	**10.87%**	35.52%	50.47%	0.50M	7.02%	47.37%	46.49%	0.47%	10.86%	38.60%	0.39M
Qwen3-Coder-480B	**28.07%**	**100.00%**	**82.46%**	9.03%	**40.12%**	54.39%	0.45M	**37.72%**	**92.11%**	**84.21%**	10.96%	**50.45%**	63.16%	2.91M
Closed-Source Model														
GPT-4o	13.16%	**100.00%**	67.54%	7.97%	22.32%	47.37%	0.11M	5.26 %	63.16%	57.89%	0.45%	6.76%	34.21%	0.58M
Gemini-2.5-Pro	12.28%	**100.00%**	74.56%	6.22%	20.55%	48.25%	0.29M	0.00%	54.39%	61.40%	0.01%	0.29%	0.00%	0.47M
Claude-4-Sonnet	**28.07%**	**100.00%**	79.82%	8.47%	38.48%	**57.89%**	0.33M	25.44%	68.42%	69.30%	11.25%	36.48%	**67.54%**	2.32M
GPT-5-mini	26.32%	98.25%	**82.46%**	10.63%	37.82%	53.98%	0.21M	36.84%	87.72%	80.70%	**12.61%**	49.64%	64.60%	1.69M
Average	20.94%	98.80%	76.97%	8.74%	31.25%	51.56%	0.29M	16.45%	67.98%	62.06%	4.92%	23.35%	46.23%	1.15M

Table 3: Performance comparison across different LLMs on NoCode-bench Full with Agentless.

Model	Success%	Applied%	RT%	FV-Micro	FV-Macro	File%	#Token
Qwen3-235B	6.62%	99.37%	75.24%	10.78%	15.77%	39.46%	0.19M
DeepSeek-v3	11.83%	**99.84%**	77.44%	**16.02%**	24.09%	**47.48%**	0.31M
DeepSeek-R1	**14.83%**	95.90%	**79.34%**	10.53%	**25.78%**	44.89%	0.57M
GPT-4o	9.31%	97.00%	71.14%	8.68%	17.81%	43.58%	0.11M
Average	10.65%	98.03%	75.79%	11.50%	20.86%	43.85%	0.30M

USD 5,124) running on NoCode-bench Full, we focus on NoCode-bench Verified for those models.

Scaffold Selection. Due to the complexity of NL-driven feature addition tasks, directly prompting LLMs to generate the required changes is impractical. Therefore, we adopt two state-of-the-art general scaffolds for solving software engineering tasks: a pipeline-based method, Agentless (Xia et al., 2025), and an agent-based method, OpenHands (Wang et al., 2025). They respectively represent two distinct paradigms to automate SE tasks using LLMs, and both have attained top scores on SWE-bench, detailed in Appendix B.2.

Evaluation Metrics. Following prior studies (Jimenez et al., 2024; Zan et al., 2025; Guo et al., 2025b), our end-to-end evaluation primarily considers the success rate (i.e, the proportion of instances for which given patch passes the accompanied test suite) of feature addition tasks (**Success%**), the success rate of patch application (**Applied%**), and the average token cost (**#Token**). Additionally, two other metrics are used: **File Matched Rate (File%)** measures alignment between model-edited and reference files, and **Regression Tests Pass Rate (RT%)** assesses the preservation of existing functionality. We introduce a new metric named **Feature Validation Rate (FV)** to evaluate whether the added feature passes the F→P tests, which is reported at both micro and macro levels to reflect overall test-level performance and instance-level robustness, respectively. Detailed definitions are provided in Appendix B.1.

5 Results

5.1 Performance on NoCode-bench Verified

The evaluation results are shown in Table 2. We find that even the most advanced LLMs still exhibit limited performance in NL feature addition. Specifically, among the evaluated models, Qwen3-Coder-480B achieves the highest success rate of 37.72% on NoCode-bench Verified with the OpenHands scaffold. The closed-source models GPT-4o and Gemini-2.5-Pro exhibit lower scores, with success rates of 5.26% and 0%. These results highlight significant performance differences across models of varying sizes and architectures under the same scaffold. Notably, the success rate on NoCode-bench Verified is much lower than that on SWE-bench Verified. For example, Claude-4-Sonnet combined with OpenHands achieves 70.4% on SWE-bench Verified but only 25.44% on NoCode-bench Verified, implying that NL-driven feature addition tasks present a substantially greater challenge compared to issue-solving tasks. A detailed comparison of scaffold performance and further analysis can be found in Appendix C.1. We further discuss the unique challenges associated with NL-driven feature addition in Section 5.3.

5.2 Performance on NoCode-bench Full

Considering the better average performance of Agentless and the high cost of OpenHands, we use Agentless to evaluate the models on NoCode-bench Full, and report the results in Table 3. We find that all four models show decreased performance on NoCode-bench Full compared to NoCode-bench Verified. Specifically, the average success rate of the four used models drops from 18.20% to 10.65%, and the file matched rate drops from 49.49% to

Table 4: Performance across different LLMs on NoCode-bench for *single-file* and *multi-file* tasks.

Model	Scaffold	Single-File Modification		Multi-File Modification	
		Success%	Applied%	Success%	Applied%
NoCode-bench Verified					
Total Instances		61		53	
Qwen3-235B	Agentless	21.31% (13)	**100.00% (61)**	3.77% (2)	**100.00% (53)**
	OpenHands	13.11% (8)	65.57% (40)	1.89% (1)	64.15% (34)
DeepSeek-v3	Agentless	31.15% (19)	**100.00% (61)**	9.43% (5)	96.23% (51)
	OpenHands	14.75% (9)	67.21% (41)	7.55% (4)	64.15% (34)
DeepSeek-R1	Agentless	40.98% (25)	98.36% (60)	7.55% (4)	88.68% (47)
	OpenHands	11.48% (7)	47.54% (29)	1.89% (1)	47.17% (25)
Qwen3-Coder-480B	Agentless	40.98% (25)	**100.00% (61)**	13.21% (7)	**100.00% (53)**
	OpenHands	**52.46% (32)**	93.44% (57)	**20.75% (11)**	90.57% (48)
GPT-4o	Agentless	19.67% (12)	**100.00% (61)**	5.66% (3)	**100.00% (53)**
	OpenHands	6.56% (4)	68.85% (42)	3.77% (2)	56.60% (30)
Gemini-2.5-Pro	Agentless	19.67% (12)	**100.00% (61)**	3.77% (2)	**100.00% (53)**
	OpenHands	0.00% (0)	52.46% (32)	0.00% (0)	56.60% (30)
Claude-4-Sonnet	Agentless	44.26% (27)	**100.00% (61)**	9.43% (5)	**100.00% (53)**
	OpenHands	39.34% (24)	68.85% (42)	9.43% (5)	67.92% (36)
GPT-5-mini	Agentless	39.34% (24)	96.72% (59)	11.32% (6)	**100.00% (53)**
	OpenHands	**52.46% (32)**	91.80% (56)	18.87% (10)	83.02% (44)
NoCode-bench Full					
Total Instances		290		344	
Qwen3-235B	Agentless	11.38% (33)	99.66% (289)	2.62% (9)	99.13% (341)
DeepSeek-v3	Agentless	20.00% (58)	**100.00% (290)**	4.94% (17)	**99.71% (343)**
DeepSeek-R1	Agentless	**24.83% (72)**	96.55% (280)	**6.40% (22)**	95.35% (328)
GPT-4o	Agentless	15.17% (44)	98.97% (287)	4.36% (15)	95.35% (328)

43.85%. This drop may be primarily due to the increased scale and complexity, as shown in Section 3, as well as the presence of some noise (e.g., ambiguous document changes or overly specific tests) in NoCode-bench Full. These factors make it more challenging. A detailed analysis of this phenomenon is provided in Appendix C.2. It is worth noting that the ranking and relative performance of the models on NoCode-bench Verified are preserved on NoCode-bench Full except Qwen3-235B and GPT-4o. A larger decline of Qwen3-235B than GPT-4o suggests that model robustness can vary as the evaluation shifts from a curated, high-quality subset to a larger set in real-world settings. We further explain this phenomenon in Appendix C.2.

5.3 Failure Analysis

To investigate why LLMs fail to solve tasks on NoCode-bench, we manually examine 160 failed instances (randomly sample 10 instances per model per scaffold) and identify the following primary reasons for LLM failures on NoCode-bench.

Lack of cross-file editing capability. In NoCode-bench Verified and NoCode-bench Full, only 53.5% and 45.7% of the instances can be solved by editing a single file, respectively, indicating that NoCode-bench requires the LLM to be good at performing cross-file edits. To further investigate how cross-file editing capability affects performance on feature addition tasks, we compute the number of modified files in the golden patch for each instance and divide the dataset into two groups: *single-file* (#modified_files = 1) and *multi-file* (#modified_files > 1). We analyze the performance of LLMs on *single-file* and *multi-file* tasks, and present the results in Table 4. On NoCode-bench Verified and NoCode-bench Full, the best-performing model solves only 20.75% and 6.40% of the *multi-file* instances, respectively. These results indicate that the lack of cross-file editing capability is one of the main reasons for LLM failures on NoCode-bench.

Lack of comprehensive understanding of existing code modules. Through manual inspection, we find that LLMs often attempt to implement new features by directly modifying the code of existing features, and 39.38% of the examined instances exhibit this error. This can lead to the new feature overriding existing features, resulting in failure in regression tests. A typical example is the patch generated by DeepSeek-v3+Agentless on the task "*pylint-7869*" (detailed in Appendix C.3). This implies that LLMs lack a comprehensive understanding of existing code modules and face challenges in figuring out what and how existing code modules should be edited.

Parsing failure in LLM's tool-calling. This type of error occurs during the tool-calling phase of the OpenHands CodeAct Agent. For Gemini-2.5-Pro, all the examined instances exhibit this error. OpenHands relies on a set of predefined tools that allow the model to access the repository, requiring the LLM to return detailed tool-call information in a strictly defined JSON format. This makes the system highly dependent on the output formatting capability of LLMs. However, nearly all LLMs fail to invoke tools in the correct output format for some samples, with Gemini-2.5-Pro being particularly prone to this issue, which directly results in a success rate of 0 on NoCode-bench Verified. A typical case is shown in Appendix C.4, where Gemini repeatedly fails to format tool calls correctly, eventually causing the context window to overflow and preventing successful patch generation. This case demonstrates that the model's capability to generate formatted output during the tool-calling significantly impacts the agent's ability to operate on the codebase and can even directly lead to failure.

6 Related Work

6.1 Benchmarks for Evaluating LLMs in SE

Evaluating the capabilities of LLMs in software engineering (SE) has attracted increasing attention, leading to the development of several benchmarks. Among these, SWE-bench (Jimenez et al., 2024) has emerged as one of the most influential benchmarks for assessing LLMs on issue resolution tasks. A line of follow-up work has extended SWE-bench to multilingual (Zan et al., 2024, 2025), multimodal (Yang et al., 2025b; Guo et al., 2025b), and industrial (Rondon et al., 2025) scenarios, promoting broader language and scenario coverage in issue-solving tasks. FEA-Bench (Li et al., 2025b) aims to evaluate LLMs on feature addition tasks, with detailed code-level specifications, docstrings, and pull request descriptions as task inputs. However, these benchmarks rely on developer-centric inputs, such as issue reports, and assume familiarity with the codebase and the implementation-level terminology (Chaparro et al., 2019; GitHub, Inc., 2025a). As a result, they provide limited insight into LLMs' capabilities in user-centric scenarios. Also, most of them focus on bug fixing or issue solving, while largely overlooking feature addition, which accounts for about 60% of software maintenance activities (Rahman, 2019).

Table 5 provides a detailed comparison between NoCode-bench and related benchmarks, highlighting several distinctive features of NoCode-bench: (1) NoCode-bench adopts a more user-centric no-code setting where user-facing documentation changes are provided as inputs, rather than code-level specifications; (2) NoCode-bench does not constrain instances to those introducing new components, recognizing that many feature additions extend existing code; (3) NoCode-bench identifies feature-adding PRs based on developer-maintained release notes to reduce noise (Xu et al., 2024; Zhang et al., 2025). In addition, we manually construct a verified subset to enable lightweight yet reliable evaluation.

6.2 Automating SE tasks with LLMs

Recently, LLM-based methods (Jiang et al., 2025) have been proposed for automating SE tasks, mainly including pipeline-based and agent-based methods. Pipeline-based methods typically decompose tasks into stages (e.g., localization, repair, and validation) based on prior knowledge and require LLMs to follow a fixed workflow. Examples include Agentless (Xia et al., 2025), and PatchPilot (Li et al., 2025a). Agent-based methods equip agents with various tools (e.g., bash and language servers) to access and modify the codebase, and rely on the decision-making and tool invocation capabilities of LLMs. Examples include OpenHands (Wang et al., 2025), RepairAgent (Bouzenia et al., 2025), and ExecutionAgent (Bouzenia and Pradel, 2025). While these methods have achieved significant advances, they are typically evaluated in a developer-centric way, where the inputs include implementation-level details and assume programming expertise. In contrast, feature addition tasks in NoCode-bench are specified with user-facing documentation changes. This mismatch introduces new and unique challenges for applying prior methods to NoCode-bench.

Table 5: Comparison with existing benchmarks.

Aspect	SWE-bench	GITS-Eval	FEA-Bench	NoCode-bench
Development scenario	Developer-centric	Developer-centric	Developer-centric	User-centric
Task type	Issue resolution	Issue resolution	Feature addition	Feature addition
Task input	Issue description	Issue description	Code-level specs, PR description	Documentation changes
Instance selection	N/A	N/A	PRs introducing new components	PRs containing doc changes
Feature identification	N/A	N/A	Prompting LLMs	Mining release notes
Verified subset	Yes	Yes	No	Yes
Open-source	Yes	No	Yes	Yes

7 CONCLUSION

We introduce NoCode-bench, a challenging benchmark for evaluating LLMs' software engineering capabilities in NL-driven feature addition. NoCode-bench is constructed with a systematic five-phase pipeline. It consists of 634 real-world feature addition tasks identified from developer-maintained release notes. Given documentation changes within a repository as input, NoCode-bench requires systems to generate a patch that adds a new feature. We evaluate 8 advanced LLMs using two representative scaffolds, Agentless and OpenHands. Our results show that the best-performing model succeeds in only 37.72% of instances in NoCode-bench Verified and 14.83% in NoCode-bench Full. This demonstrates that NL-driven feature addition is a uniquely difficult and unsolved problem. Moreover, we analyze failure cases on NoCode-bench to guide future improvements in cross-file editing, code module understanding, and tool-calling capabilities.

LIMITATIONS

NoCode-bench, while designed to reflect realistic documentation-driven feature additions, has several limitations.

First, the benchmark is constructed from historical GitHub repositories, raising the possibility of data leakage, as such data may appear in the training corpora of modern large language models. We mitigate this risk by masking PR numbers and other metadata, and the consistently low success rates across strong LLMs suggest limited practical impact, though the risk cannot be completely ruled out.

Second, the current benchmark focuses on well-maintained open-source Python projects with structured release notes. This choice ensures clean and interpretable task definitions but limits language and ecosystem diversity. Although the construction methodology is language-agnostic and can be extended to other settings, future work is needed to broaden project coverage.

In addition, our evaluation relies on two scaffolds (i.e., Agentless and OpenHands), which represent strong baselines but do not cover the full spectrum of possible code-execution or agent frameworks. Performance under other scaffolds remains unexplored. Nonetheless, the modest results observed across multiple LLMs indicate that NoCode-bench poses meaningful challenges that extend beyond scaffold-specific behaviors.

Ethics Statement

NoCode-bench is constructed entirely from publicly available open-source software projects, all of which are used in accordance with their original licenses. The benchmark is intended for research purposes and does not involve any sensitive personal data. Both the dataset and the accompanying code will be released for academic use. Users should be aware that code produced by models when performing benchmark tasks could potentially be harmful to computer systems, and we recommend evaluating model outputs in controlled environments such as Docker containers.

References

Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1199–1210.

Anthropic. 2024. Claude 4. https://www.anthropic.com/news/claude-4. Accessed: 2025-01.

Tingting Bi, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. 2020. An empirical study of release note production and usage in practice. *IEEE Transactions on Software Engineering*, 48(6):1834–1852.

Tegawendé F. Bissyandé, David Lo, Lingxiao Jiang, Laurent Réveillère, Jacques Klein, and Yves Le Traon. 2013. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 188–197.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An autonomous, LLM-based agent for program repair. In *International Conference on Software Engineering (ICSE)*.

Islem Bouzenia and Michael Pradel. 2025. You name it, i run it: An llm agent to execute tests of arbitrary projects. In *Proceedings of the 34rd ACM SIGSOFT International Symposium on Software Testing and Analysis*.

Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. 2019. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 86–96.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. Introducing SWE-bench verified.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

GitHub, Inc. 2025a. GitHub Issues · Project planning for developers. https://github.com/features/issues.

GitHub, Inc. 2025b. GitHub Spec Kit: Toolkit to help you get started with Spec-Driven Development. https://github.com/github/spec-kit.

R.L. Glass. 2001. Frequently forgotten fundamental facts about software engineering. *IEEE Software*, 18(3):112–111.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025a. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Lianghong Guo, Wei Tao, Runhan Jiang, Yanlin Wang, Jiachi Chen, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. 2025b. Omnigirl: A multilingual and multimodal benchmark for github issue resolution. In *Proceedings of the 34rd ACM SIGSOFT International Symposium on Software Testing and Analysis*.

Martin Hirzel. 2023. Low-code programming models. *Communications of the ACM*, 66(10):76–85.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79.

Zhonghao Jiang, David Lo, and Zhongxin Liu. 2025. Agentic software issue resolution with large language models: A survey. *arXiv preprint arXiv:2512.22256*.

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*.

Yulia Kumar, Israel Akinwunmi, and Dov Kruger. 2025. Evaluating the advantage of an ai-native ide cursor on programmer performance. In *2025 IEEE Integrated STEM Education Conference (ISEC)*, pages 1–8.

Hongwei Li, Yuheng Tang, Shiqi Wang, and Wenbo Guo. 2025a. Patchpilot: A cost-efficient software engineering agent with early attempts on formal verification. In *Forty-second International Conference on Machine Learning*.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. 2025b. FEA-bench: A benchmark for evaluating repository-level code generation for feature implementation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

OpenAI. 2024. Gpt-4o system card. *Preprint*, arXiv:2410.21276.

OpenAI. 2025. Introducing gpt-5. https://openai.com/index/introducing-gpt-5/. Accessed: 2025-10-17.

OpenAPI. 2025. Usage | openapi generator. https://openapi-generator.tech/docs/usage.

Pallets. 2025. Flask: Micro framework for building web applications. https://flask.palletsprojects.com/en/stable/.

Tom Preston-Werner. 2010. Readme driven development. https://tom.preston-werner.com/2010/08/23/readme-driven-development.

Rohith Pudari and Neil A Ernst. 2023. From copilot to pilot: Towards ai supported software development. *arXiv preprint arXiv:2303.04142*.

Mohammad Masudur Rahman. 2019. Supporting code search with context-aware, analytics-driven, effective query reformulation. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 226–229.

Nikitha Rao, Jason Tsay, Kiran Kate, Vincent Hellendoorn, and Martin Hirzel. 2024. Ai for low-code for ai. In *Proceedings of the 29th International Conference on Intelligent User Interfaces*, pages 837–852.

Damien Rolon-Mérette, Matt Ross, Thaddé Rolon-Mérette, and Kinsey Church. 2016. Introduction to anaconda and python: Installation and setup. *Quant. Methods Psychol*, 16(5):S3–S11.

Pat Rondon, Renyao Wei, José Cambronero, Jürgen Cito, Aaron Sun, Siddhant Sanyam, Michele Tufano, and Satish Chandra. 2025. Evaluating agent-based program repair at google. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. 2020. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178.

Mozhan Soltani, Felienne Hermans, and Thomas Bäck. 2020. The significance of bug report elements. *Empirical Software Engineering*, 25(6):5255–5294.

SymPy Development Team. 2025. Sympy: Python library for symbolic mathematics. https://docs.sympy.org/latest/index.html.

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, and 5 others. 2025. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10.

Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. 2024. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

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 I. Wang, and Ofir Press. 2025b. SWE-bench multimodal: Do ai systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations*.

John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025c. SWE-smith: Scaling data for software engineering agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Liguo Yu. 2009. Mining change logs and release notes to understand software maintenance and evolution. *CLEI Electronic Journal*, 12(2):1–1.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, and 1 others. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*.

Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, and 1 others. 2024. Swe-bench-java: A github issue resolving benchmark for java. *arXiv preprint arXiv:2408.14354*.

Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, and 1 others. 2025. Siren's song in the ai ocean: A survey on hallucination in large language models. *Computational Linguistics*.

A BENCHMARK DETAILS

A.1 Construction Details

This appendix provides implementation-level details of the NoCode-bench construction pipeline.

A.1.1 Project and Release Note Processing

While the main text describes the project selection criteria, here we detail how release notes are processed in practice. We treat each feature-labeled item in a release note as an *update entry*. Because release note formats vary substantially across projects (e.g., Markdown, HTML, or Jupyter Notebook), we implement lightweight, project-specific parsers to normalize release note structures. These parsers extract: (1) the textual description of each feature entry, and (2) references to the corresponding GitHub PRs. This information will be used for subsequent crawling and filtering.

As illustrated in Phase 1 of Figure 4, we can locate the release notes on the project's homepage. The change log for each version contains clearly labeled feature entries, each accompanied by a link to the corresponding PR. We extracted the text part (i.e., Added the objects.Text stat) and PR number (i.e., #3051) from this example.

A.1.2 PR Crawling and Metadata Extraction

For each PR linked from release notes, we automatically crawl the full PR metadata using the GitHub API. We extract the base commit (i.e., the commit on which the feature branch was originally developed) by identifying the merge base between the PR branch and the main branch at merge time. This base commit is used to reconstruct the pre-feature codebase. To construct benchmark inputs and evaluation oracles, we compute three diffs for each retained PR: (1) the documentation diff, which serves as the task input; (2) the feature patch, which contains source code changes; and (3) the test patch, which is used to verify the correctness of the feature implementation.

A.1.3 Environment Reconstruction and Dependency Repair

This section details the environment construction process summarized in the main text. For each project, we construct a single base Docker image and manage instance-level version isolation using Anaconda environments. Instances are first grouped by minor versions using automated scripts, which parse version-related files such as *setup.py*, *__init__.py*, and release notes. We then manually inspect the format and location of environment configuration files, such as *README.md*, *requirements.txt*, *pyproject.toml*, or *pom.xml* to identify how dependencies are declared. Automated scripts are developed to extract and install these dependen-

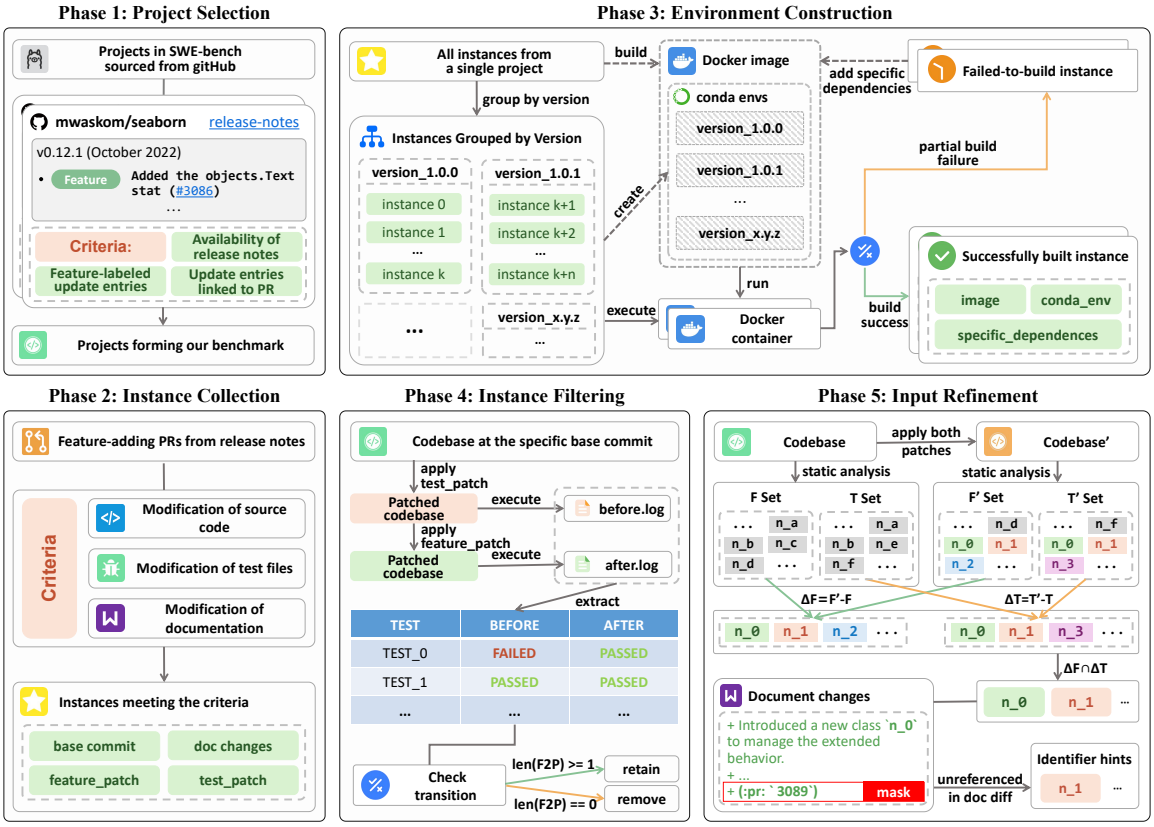

Figure 4: The workflow of building our benchmark.

cies accordingly. Some instances fail to build or execute due to loosely pinned or outdated dependencies. For such cases, we log build and runtime errors and apply minimal manual fixes, such as pinning a dependency version or adding a missing library. Each fix is recorded as a reproducible shell script. Because instances within the same project often share similar dependency issues, a single fix typically applies to many instances, resulting in limited additional human effort.

A.1.4 Test Execution and Status Extraction

To verify each instance as a valid feature addition, we execute all test files that are modified or added in the *test_patch*. For each instance, we collect two sets of logs, i.e., *before.log* and *after.log*. *before.log* is obtained by applying the *test_patch* to the base commit and then executing the corresponding test suite, while *after.log* is obtained by applying both the *test_patch* and the *feature_patch* to the base commit, followed by test execution.

We support multiple test frameworks (e.g., *pytest*, *unittest*) by normalizing their outputs into a unified representation of per-test execution status. Test case identifiers and their corresponding outcomes are extracted using framework-specific parsers. These normalized logs are then used to determine test status transitions, which serve as the basis for instance filtering as described in the main text.

A.1.5 Identifier Hint Extraction

This section provides implementation details for identifier hint extraction. Given an instance, we first switch the codebase to its base commit. We then identify the list of files modified by the *feature_patch* and the *test_patch*, denoted as *feature files* and *test files*, respectively. For each file list, we extract all file names and the identifiers (e.g., classes, functions, attributes) in these files through static analysis, producing a set of entities. We refer to the entity sets extracted from feature files and test files as F and T, respectively. Next, we apply both *feature_patch* and *test_patch* to the codebase and repeat the same process to obtain the post-patch entity sets, denoted F' and T'. We compute the differences between the pre- and post-patch sets to obtain the sets of newly introduced entities in feature files and test files, respectively, i.e., $\Delta_F = F' - F$ and $\Delta_T = T' - T$. We then compute the intersection of Δ_F and Δ_T to isolate entities that are newly introduced and referenced in both test and

Table 6: Guidelines for the accuracy of the evaluation.

Score	Accuracy of the Evaluation
0	The tests perfectly cover all aspects of the new feature mentioned in the doc changes.
1	The tests cover the majority of aspects of the feature involved in doc changes, but may overlook some special cases.
2	The tests are too narrow/broad, only test a few special cases, or test content far beyond the description in the document.
3	The tests look for something different from what the doc changes is about.

implementation code. This step ensures we only consider entities relevant to the verification process. Finally, we remove all entities that are already mentioned in the documentation changes, yielding the final set of undocumented but test-referenced entity names.

A.2 NoCode-bench Verified

A.2.1 Sampling and Annotation Protocol

To ensure project diversity, we sample up to 35 task instances per project (or fewer if insufficient candidates exist), yielding a total of 238 candidate tasks. Considering the subjectivity of human judgment, each sample is annotated independently by two different annotators. For quality assurance, we conservatively select the more stringent label when disagreements arise, following SWE-bench Verified. As a result, each annotator evaluated approximately 95 task instances. We construct NoCode-bench Verified by randomly sampling candidate instances from NoCode-bench Full. To ensure diversity across projects, we cap the number of sampled instances at 35 per project (or fewer if insufficient candidates exist), resulting in a total of 238 candidate tasks.

Each candidate task is independently annotated by two annotators. All annotators have at least three years of Python development experience and hold a bachelor's degree or higher in a relevant field. In cases of disagreement, we conservatively adopt the stricter label, following the protocol of SWE-bench Verified (Chowdhury et al., 2024). Under this setup, each annotator evaluates approximately 95 tasks.

A.2.2 Annotation Rubric

Clarity of the Task. Annotators assess whether the documentation change alone provides a sufficiently clear and unambiguous specification of the intended feature.Annotators act as experienced software engineers with full access to the codebase and judge whether a reasonable implementation can be inferred from the documentation change.

Accuracy of the Evaluation. Annotators assess whether the test patch provides appropriate and targeted functional coverage of the behavior described in the documentation change. Unlike SWE-bench Verified, which focuses on whether tests can reliably evaluate diverse correct implementations (e.g., avoiding false negatives due to mismatches in naming or structure), we mitigate such concerns during benchmark construction in Phase 5. As a result, our evaluation emphasizes whether the provided test cases faithfully and comprehensively cover the functionality described in the documentation changes. Scoring is based on the criteria in Table 6.

Additionally, if the annotator believes that there are other major problems with the sample that have not been considered, i.e., any other reasons this sample should not be used in our setup for evaluating coding ability, the annotator needs to flag the sample and provide a basic explanation.

Table 7: Human verification results for task clarity and test accuracy. For the *Other Problem* column, a score of 0 indicates no issues were identified, while a score of 3 indicates the presence of significant issues.

Score	Task Clarity	Test Accuracy	Other Problem
0 (best)	59	68	220
1	99	94	–
2	59	64	–
3 (worst)	18	9	18

A.2.3 Additional Filtering Details

After all the samples have been annotated, we filter out any sample from the original test set where the severity of the clarity of the task or the accuracy of the evaluation is labeled as 2 or above. We also filter out all samples that have a major problem flagged by annotators. For example, in *pytest-3576*, the associated PR includes a large number of unrelated documentation changes, which are not indicative of genuine feature additions and thus do not meet the criteria of our benchmark. As summarized in Table 7, a total of 106 instances were removed due to unclear task descriptions or inaccurate evaluations, and 18 instances were excluded due to other major problems.

B DETAILED EXPERIMENT SETUP

B.1 Evaluation Metrics

- **File Matched Rate (File%):** The percentage of modified files in the submitted patch that match one of the file paths edited by the ground truth patch.
- **Regression Tests Pass Rate (RT%):** The proportion of tasks where all regression tests pass after applying the generated patch.
- **Feature Validation (FV):** This metric aims to evaluate the completeness of the LLM's implementation of a specific feature, which we measure based on the pass rate of F→P tests. Specifically, we calculate FV from both macro and micro perspectives.
  - **Micro-level FV (FV-Micro):** The overall ratio of passed F→P tests to the total number of F→P tests across all instances: FV-Micro $= \frac{\sum_{i=1}^{N} \#\text{Passed}_i}{\sum_{i=1}^{N} \#\text{Total}_i}$, where $\#\text{Passed}_i$ and $\#\text{Total}_i$ denote the number of passed and total F→P tests for instance i, respectively.
  - **Macro-level FV (FV-Macro):** The average F→P pass rate across instances: FV-Macro $= \frac{1}{N}\sum_{i=1}^{N} \frac{\#\text{Passed}_i}{\#\text{Total}_i}$, where N is the total number of instances.

B.2 Used Scaffolds

- **Agentless** adopts a hierarchical approach to sequentially identify the relevant files, classes or methods, and lines of code that require modification, and then generates patches based on the localized content with LLM. Following Guo et. al. (Guo et al., 2025b), we use Agentless-1.0 for evaluation.
- **OpenHands** is a widely adopted platform for building software development agents. We adapt the document changes in NoCode-bench as inputs to the OpenHands CodeAct Agent and utilize the standard evaluation docker images provided by NoCode-bench as a sandbox environment to ensure the correct execution of tasks.

C FURTHER ANALYSIS OF EXPERIMENT RESULTS

C.1 Detailed Analysis of NoCode-bench Verified Results

Performance Across Scaffolds. The Agentless scaffold achieves an average patch application rate of 98.80%, whereas OpenHands produces applicable patches for only 67.98% of the tasks on average. For regression tests, 76.97% of the patches generated by Agentless passed all regression tests on average, compared to only 62.06% for OpenHands. In terms of full F2P test suite pass rate, Agentless achieves 8.74% (FV-Micro), with an average of 31.25% (FV-Macro) of F2P tests passed per instance, while OpenHands achieves 4.92% and 23.35%, respectively. The results show that Agentless outperforms OpenHands on NoCode-bench Verified.

Why Agentless Outperforms OpenHands. Through manual inspection, we find that the reason lies in how OpenHands maintains a dialogue history to record interactions between the agent and the repository. Since solving tasks in NoCode-bench Verified requires editing an average of 2.39 files, the observed context often exceeds the model's context window, resulting in a large number of incomplete or empty patches. In contrast, Agentless uses hierarchical localization to limit the length of retrieved context and applies post-processing to verify patch completeness, leading to better overall performance.

Gemini-2.5-Pro's Complete Failure on OpenHands. By inspecting the failure cases, we find that Gemini-2.5-Pro is almost unable to generate correct invocation formats for any of the tools used by OpenHands. This directly prevents it from accessing and editing the repository.

C.2 Detailed Analysis of NoCode-bench Full Results

Why Performance Drops on NoCode-bench Full. Since NoCode-bench Full is a large-scale dataset involving more extensive file modifications and more complex code changes, it presents a diverse range of challenges. In addition, NoCode-bench Full may contain some noise, including unclear documentation changes or low-coverage regression tests, which make it more representative of real-world, NL feature addition tasks, but harder for LLMs to solve.

Model Robustness Comparison. For DeepSeek-R1, DeepSeek-v3 and Qwen3-235B, success rates drop from 25.44% to 14.83%, 21.05% to 11.83%, and 13.16% to 6.62%, respectively, while GPT-4o only drops from 13.16% to 9.31%. Qwen3-235B and GPT-4o achieve similar performance

```
"""DeepSeek-v3 Patch"""
diff --git a/pylint/reporters/text.py b/pylint/reporters/text.py
--- a/pylint/reporters/text.py
+++ b/pylint/reporters/text.py
@@ -294,14 +295,15 @@ class ColorizedTextReporter(TextReporter):
     """Manage message of different types, and colorize output
     using ANSI escape codes.
     """
-    if msg.module not in self._modules:
-        msg_style = self._get_decoration("S")
-        if msg.module:
-            modsep = colorize_ansi(f"************* Module {msg.module}", msg_style)
-        else:
-            modsep = colorize_ansi(f"************* {msg.module}", msg_style)
-        self.writeln(modsep)
-        self._modules.add(msg.module)
+    if getattr(self.linter.config, "output_format", None) != "no-header":
+        if msg.module not in self._modules:
+            msg_style = self._get_decoration("S")
+            if msg.module:
+                modsep = colorize_ansi(f"************* Module {msg.module}", msg_style)
+            else:
+                modsep = colorize_ansi(f"************* {msg.module}", msg_style)
+            self.writeln(modsep)
+            self._modules.add(msg.module)
     msg_style = self._get_decoration(msg.C)
```

```
"""Golden Patch"""
diff --git a/django/forms/models.py b/django/forms/models.py
--- a/django/forms/models.py
+++ b/django/forms/models.py
@@ -212,6 +213,18 @@ def _display(self, layout: Section) -> None:
         TextWriter().format(layout, self.out)
+class NoHeaderReporter(TextReporter):
+    """Reports messages and layouts in plain text without a module header."""
+
+    name = "no-header"
+
+    def handle_message(self, msg: Message) -> None:
+        """Write message(s) without module header."""
+        if msg.module not in self._modules:
+            self._modules.add(msg.module)
+        self.write_message(msg)
 class ParseableTextReporter(TextReporter):
     """A reporter very similar to TextReporter, but display messages in a form
     recognized by most text editors :
```

Figure 5: A failure example of *pylint-dev__pylint-7869* caused by the LLM's lack of architectural understanding.

on NoCode-bench Verified, while GPT-4o outperforms Qwen3-235B on NoCode-bench Full. These results demonstrate that compared to DeepSeek-v3 and Qwen3-235B, GPT-4o performs more robustly in the scenarios that are closer to real-world conditions.

C.3 Case Study: Lacking Understanding of Existing Code Modules in pylint-7869

Figure 5 highlights a critical difference between the patch generated by DeepSeek and the golden patch for the task "pylint-7869". This task requires introducing a new "no-header" option to Pylint. When this option is enabled, Pylint's output will omit (i.e., not display) the header information that normally shows the name of the module being analyzed. DeepSeek-v3 directly modifies the member functions of the *ColorizedTextReporter* class to implement the "no-header" option, whereas the golden patch introduces a new *NoHeaderReporter* class to achieve the same goal. Compared to the golden patch, DeepSeek-v3's modification breaks the original functionality. Although it passes all F→P tests related to the new feature, it results in numerous regression test failures. This example demonstrates how LLMs may lack understanding of existing code modules needed to extend functionality without breaking existing features. Users may alleviate such phenomena by providing the LLM with information about the differences between existing modules or by guiding the LLM to analyze existing modules through a predefined plan.

C.4 Case Study: Tool-Calling Format Error in astropy__astropy-9425

Figure 6 shows a tool-calling dialogue from OpenHands while attempting to solve the task

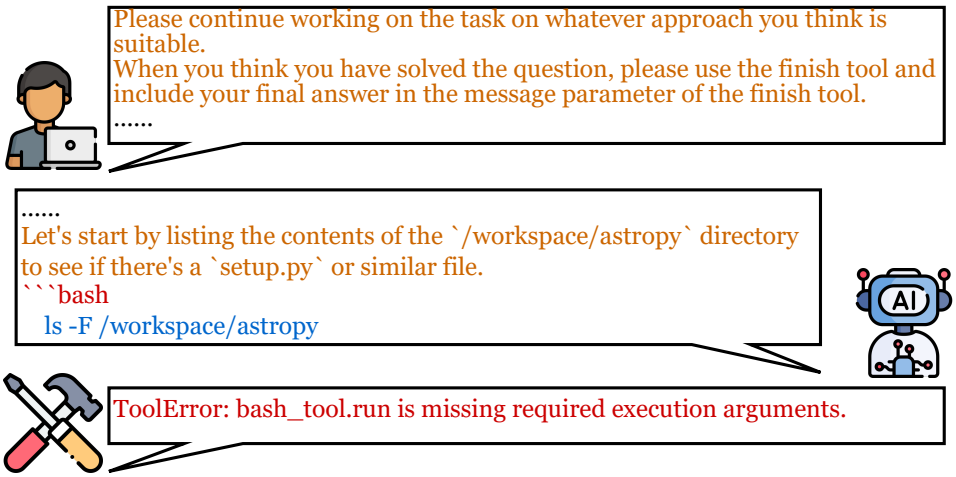

Figure 6: A failure example of *astropy__astropy-9425* caused by the LLM's incorrect format of tool calling.

astropy__astropy-9425. In the first round, the agent chooses to use the "bash" tool to inspect the repository structure. However, since the tool requires the LLM to return commands in JSON format, Gemini fails the call by responding with the intended command in plain text instead. As a result, the tool invocation fails, and the LLM repeatedly retries the same tool call. Eventually, the conversation history fills the context window, preventing OpenHands from successfully generating a patch for this instance. This example illustrates how formatting errors in tool calls can cascade into complete task failure.