NoCode-bench: A Benchmark for Evaluating Natural Language-Driven Feature Addition

Le Deng*
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, China
dengle@zju.edu.cn

Zhonghao Jiang*
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, China
zhonghao.j@zju.edu.cn

Jialun Cao
The Hong Kong University of Science and Technology
Hong Kong, China
jialuncao@ust.hk

Michael Pradel
University of Stuttgart
Stuttgart, Germany
michael@binaervarianz.de

Zhongxin Liu†
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, China
liu_zx@zju.edu.cn

Leaderboard HuggingFace GitHub DockerHub

ABSTRACT

Natural language-driven no-code development aims to enable users to specify software functionality through natural language (NL) without directly editing source code, holding the promise for democratizing software development and increasing productivity. Large language models (LLMs) have demonstrated their great potential in realizing this development paradigm. As software documentation serves as an authoritative NL specification of software functionality, we envision a natural and promising no-code development scenario where users or developers update documentation to specify new features, and LLMs automatically infer and perform the corresponding code changes. To advance no-code development, this work introduces NoCode-bench, a benchmark designed to evaluate LLMs on real-world NL-driven feature addition tasks. NoCode-bench consists of 634 tasks across 10 projects involving about 114k code changes in total, each of which pairs a user-facing documentation change and the corresponding code implementation that can be validated against developer-written test cases. It is constructed through a five-phase pipeline starting from release notes, offering broad coverage and simulating real-world development scenarios. To facilitate lightweight and reliable evaluation under limited resources, we further curate a human-validated subset named NoCode-bench Verified. It covers 114 high-quality instances 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 28.07%. In addition, our analysis reveals that LLMs face key challenges in cross-file editing, comprehensively understanding the codebase structure, and tool calling. These results indicate that current LLMs are not yet ready for this task. NoCode-bench formulates a new challenge and paves the way for future progress in NL-driven no-code software development.

*Equal contribution.
†Corresponding Author.

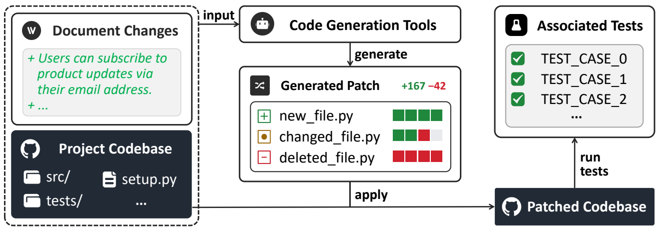


Figure 1: An overview of the no-code feature addition task.

1 INTRODUCTION

No-code development refers to a paradigm where users specify software functionality through interfaces that are natural and easy to learn, such as natural language or visual tools, rather than writing source code directly [15, 27]. Prior work has shown that no-code development holds promise for democratizing software development and significantly increasing productivity [27, 31]. Among these interfaces, natural language (NL) stands out as the most intuitive and familiar one for humans, making NL-driven no-code development attractive for users. Specifically, in such development paradigm, users specify how the software should behave or change in natural language, with the underlying code automatically generated by the no-code system [12, 41].

Recently, large language models (LLMs) demonstrate remarkable capabilities in NL-based code generation [4, 10, 22], paving a promising way for realizing NL-driven no-code development. To understand and further improve the performance of LLMs in NL-driven no-code development, we need high-quality benchmarks where the instances take NL specifications as input and can simulate real-world development tasks. We notice that user-facing software documentation usually serves as an authoritative and comprehensive NL specification of software functionality [1], and thus is suitable for being used to construct benchmarks for NL-driven no-code development.

To construct a benchmark based on documentation, one option is to formulate each task in the benchmark as generating the entire code base of a real-world project based on its user-facing documentation. However, software documentation can range from dozens to thousands of pages, and software projects often consist of tens or hundreds of thousands of lines of code, making such a benchmark too challenging to sensitively measure the progress in this field. Instead, our idea is to focus on NL-driven feature addition tasks, which aim to generate code changes based on documentation changes. Compared to developing a project from scratch, adding a feature to an existing project is finer-grained and more achievable. Moreover, feature addition is a crucial and frequent task encountered by developers in their daily work. For example, prior work [13, 26] has shown that approximately 60% of overall software development costs are allocated to maintenance activities, with feature addition accounting for around 60% of these maintenance efforts.

While several benchmarks [16, 19, 23, 39, 43], including SWE-bench [16], are proposed to evaluate LLMs on real-world software development tasks, none of them target no-code development scenarios. In addition, most of them focus primarily on bug fixing or issue resolution tasks, where issue descriptions instead of software documentation are taken as input. For example, only 7.3% of the tasks in SWE-bench involve enhancement tasks like feature additions.

To fill this gap, we present NoCode-bench, a new benchmark designed to evaluate LLMs' capability in no-code feature addition on real-world software projects. NoCode-bench consists of 634 carefully collected instances derived from high-quality open-source projects hosted on GitHub. As illustrated in Figure 1, each instance takes user-facing documentation changes as input and expects the model to generate corresponding code changes. The implementation is validated using developer-written test cases. While we assume test cases to be given in our benchmark to support objective and reproducible evaluation, we envision a future where such tests can also be generated automatically by LLMs [2, 20, 21]. To ensure the reliability and comprehensiveness of our benchmark, we develop a systematic five-phase construction pipeline that includes project selection, instance collection, environment construction, instance filtering, and input refinement. Unlike prior pipelines that mine instances directly from GitHub issues or pull request (PR) histories, our pipeline introduces a novel starting point, i.e., developer-maintained release notes. By anchoring the benchmark construction on release notes, we are able to identify developer-confirmed feature addition tasks and significantly reduce label noise. This pipeline allows us to create realistic and reproducible evaluation scenarios. 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.

NoCode-bench and its verified subset can serve as valuable resources for advancing research in LLM-based software engineering (SE), especially for no-code feature addition. We leverage NoCode-bench Verified to systematically evaluate a range of state-of-the-art LLMs with the Agentless and OpenHands frameworks. Our evaluation shows that, despite incurring high token consumption, the best success rate of existing models is only 28.07%, underscoring the substantial complexity of no-code feature addition. A closer analysis reveals three primary sources of failure: (1) lack of cross-file editing capability, where LLMs struggle to locate and modify multiple relevant files; (2) lack of comprehensive understanding of the codebase structure, leading to incorrect insertion points and broken existing functionality; and (3) insufficient tool-calling ability, where models fail to invoke appropriate tools. These failure patterns indicate that LLMs are not yet ready for NL-driven no-code feature addition and need to integrate more deeply with project-wide context, structure, and tooling.

In summary, our main contributions are:

- We introduce NoCode-bench, a novel benchmark specifically designed for evaluating no-code feature addition. NoCode-bench addresses an important gap in existing SE benchmarks and paves the way for NL-driven no-code 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 no-code development scenarios. It is composed of 634 feature addition tasks collected from popular open-source projects. Each task corresponds to a real 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 used to assess the correctness of the feature implementation. The goal of each task is to generate a patch that correctly implements the feature described in the documentation change and passes the associated tests, as illustrated in Figure 1.

This section outlines the construction process of NoCode-bench and the task definition. In addition, we describe in detail how we constructed a reliable subset, NoCode-bench Verified, through a combination of manual validation and refinement.

2.1 Benchmark Construction

We focus on high-quality, real-world software development data to construct NoCode-bench through five phases. As shown in Figure 2, we begin with carefully selected projects (Phase 1), then progressively identify, filter, and collect instances (Phase 2), reconstruct their development environments (Phase 3), filter instances based on execution (Phase 4), and finally refine the task input (Phase 5).

2.1.1 Phase 1: Project Selection. Our project selection starts with the projects used in SWE-bench [16], which are a set of actively maintained and well-documented open-source projects. Unlike SWE-bench, which constructs instances by linking GitHub issues to corresponding PRs, we identify real-world feature addition tasks by starting from release notes, which concisely highlight important changes in each software version, are written or confirmed by project developers, and thus provide authentic and accurate labels.

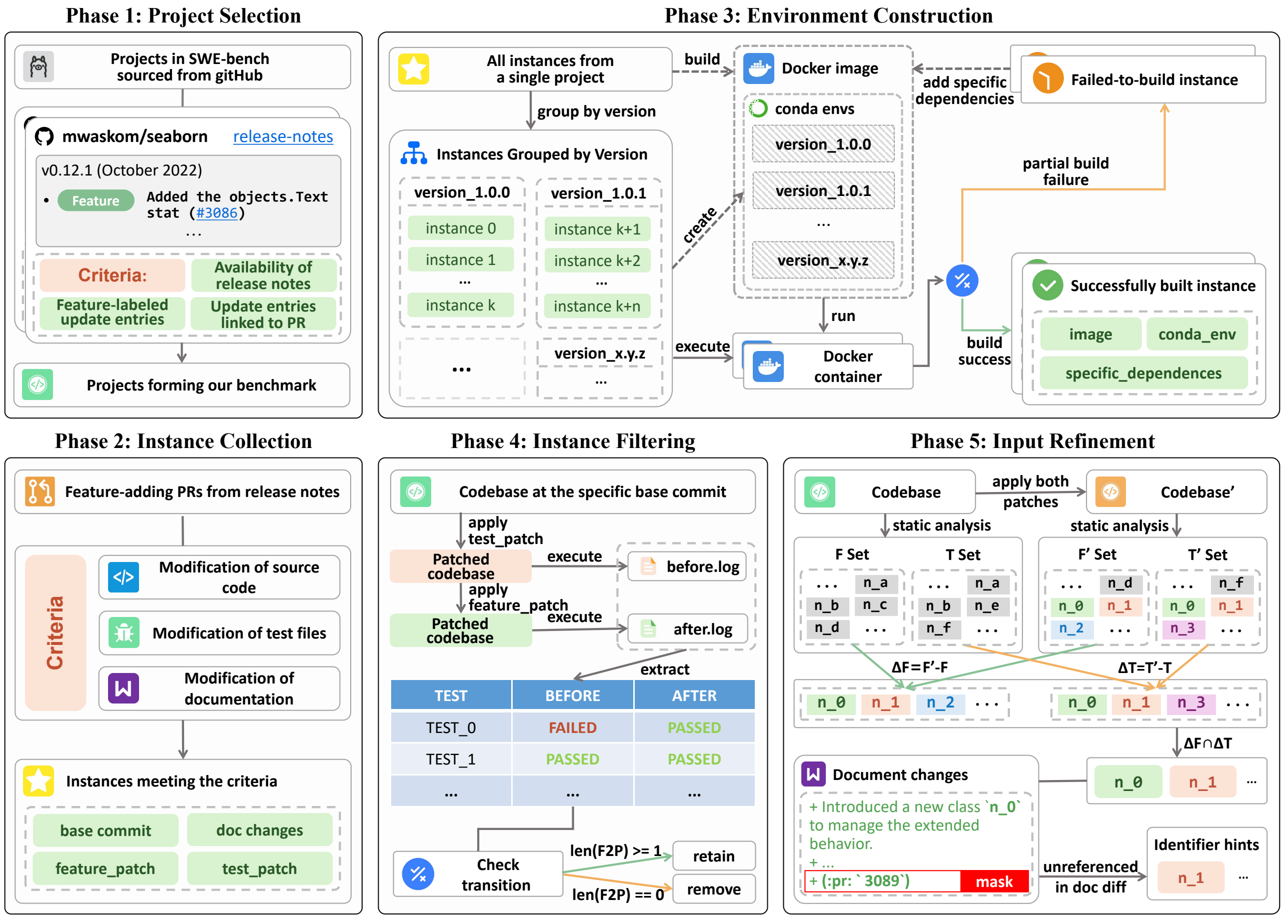


Figure 2: The workflow of building our benchmark.

Well-maintained release notes often annotate each change with its type, making them a reliable source for identifying feature additions. In this context, we refer to each change item in a release note as an *update entry*. To ensure that the constructed benchmark is reliable, representative, and suitable for evaluating no-code feature addition, we apply additional criteria when selecting projects. Specifically, we require that projects satisfy the following conditions:

- **Availability of release notes:** Prior work [6, 42] shows that a project with release notes is more likely to be well-maintained. Moreover, release notes summarize code changes introduced in different versions, and thus are useful for identifying and collecting feature-adding instances.
- **Use of feature-labeled update entries:** We define a *feature addition* as a change that introduces new user-facing capabilities or extends existing functionality. Examples include adding a new button to trigger an action or allowing export into a new file format. In contrast, changes such as bug fixes, code refactoring, or performance tuning are not considered. We require that release notes explicitly label each entry with its type (e.g., "feature", "enhancement") to support reliable instance identification.
- **Update entries linked to GitHub PRs:** These links allow us to trace the documented features back to their corresponding code changes.

As illustrated in Phase 1 of Figure 2, 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 (e.g. Feature Added the objects.Text stat (#3051)).

Following this approach, we evaluate all 12 projects from SWE-bench and find that 10 of them meet the above conditions. Projects like `flask` [25] and `sympy` [32] lack feature-labeled annotations and are thus excluded. This selection process lays a solid foundation for the subsequent phases of our benchmark development.

2.1.2 Phase 2: Instance Collection. This phase aims to crawl feature addition PRs for each project selected in Phase 1. For each selected project, we collect all PRs that correspond to the feature additions identified in Phase 1 and apply the following filtering criteria:

- **Modification of source code:** The PR must modify the source code of the project to ensure it implements the feature mentioned in the release note.
- **Modification of test files:** The PR must include changes to test files, ensuring that there are appropriate tests to verify the correctness of the feature implementation.
- **Modification of documentation:** The PR must update the project documentation, which can provide a clear description of the added feature.

Unlike SWE-bench, which filters PRs based on modifications of source code and test files, our benchmark imposes an additional constraint, i.e., modification of documentation. This aligns with the no-code feature addition scenarios, where the documentation change serves as the primary input to guide the feature addition.

Additionally, we set an August 2024 cutoff and retain only PRs merged before this date, aligning with SWE-bench for fair comparisons. After filtering, we collect detailed information for each PR, including the *base commit*, the *documentation change*, *feature_patch*, and *test_patch*. The base commit refers to the commit on which the feature was originally developed, and is used to ensure reproducible reconstruction of the pre-feature codebase. A documentation change denotes the diff containing all changes made to the project documentation, which serves as the task input. *feature_patch* refers to the code change in the PR, which implements the new functionality; *test_patch* is the test change in the PR, which is used to verify the correctness of the feature implementation. Each PR that passes the filtering process is treated as an instance in our benchmark. In total, this phase yields 1571 feature-adding instances that serve as candidates for subsequent processing.

2.1.3 *Phase 3: Environment Construction.*

Unlike SWE-bench, which constructs a separate Docker image for each instance, we adopt a more resource-efficient strategy. SWE-bench's per-instance Docker approach incurs significant storage overhead and is less scalable when instances share highly similar environments [40]. In our benchmark, many instances come from the same project and only differ slightly across minor versions, with occasional instance-specific requirements. Thus, we construct a single base Docker image per project and manage version isolation using Anaconda [3].

To ensure the correct execution and evaluation of feature addition tasks, each instance must be built and executed in an isolated environment to prevent interference. In this phase, we rely on both automated identification and manual inspection to construct the runtime environment for each instance. Specifically, for a given project, we first group instances 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 dependencies accordingly.

Despite our careful exploratory setup, some instances cannot be fully built or executed due to issues such as loosely defined dependency versions or breaking updates in libraries. For example, if a project does not strictly pin the version of a dependency, an API used in the original code may be deprecated or changed in a newer library version, causing the instance to fail. To address this, we log the build and runtime outputs of each instance, and for the failed instances, we manually fix incorrect or missing dependency specifications and record the repair process as a reproducible *shell* instruction. In practice, such manual refinement is required only occasionally. Since instances within the same project often share similar environment problems, a single fix (e.g., adding or pinning the version of a missing dependency) typically applies to many instances, resulting in minimal additional human effort.

This phase ensures that each instance is equipped with a clean and functional containerized environment, establishing a reliable foundation for subsequent testing and analysis.

2.1.4 *Phase 4: Instance Filtering.*

After collecting instances and constructing runnable environments, we proceed to verify each instance as a valid feature addition by analyzing the behavioral differences in the test execution before and after applying the feature patch. Specifically, we execute all test files that are modified or added in the *test_patch*, as they are expected to reflect the intended behavior of the new feature. For each instance, we execute the relevant test cases and 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.

From these logs, we extract the execution status of each test case. Following the method used by SWE-bench, we focus primarily on the transition of test case status between before.log and after.log. Specifically:

- We collect all test cases with a PASSED → PASSED transition as regression tests, and refer to them as *P2P tests*. These test cases ensure that the addition of the new feature does not break existing features.
- To confirm that the feature is correctly implemented, we identify test cases with a FAILED → PASSED transition as validation tests, and refer to them as *F2P tests*. These test cases validate that the added feature is now operational.

Together, these two types of test cases constitute the complete set of relevant tests for evaluating a given instance. If an instance does not have any F2P test, it is excluded from the benchmark, as there is no clear evidence that a feature has been added. After applying this filtering, we retain a total of 634 instances.

It is important to note that, unlike SWE-bench, we do not discard instances in which before.log contains errors such as *ImportError* or *AttributeError*. These errors are often expected in feature addition scenarios because the new feature may introduce previously undefined modules, classes, or functions. As a result, test cases referring to these components will naturally fail before the feature patch is applied, and the corresponding instances should not be filtered when collecting feature addition tasks.

2.1.5 *Phase 5: Input Refinement.*

To further improve the accuracy and reliability of model evaluation, we introduce a final phase to address a critical issue observed during benchmark construction, namely, missing but essential entity names in the task input.

During feature addition, it is common for developers to introduce new program entities such as files, classes, methods, or attributes. However, we observed that in some instances, the names of these newly introduced entities appear explicitly in test cases and implementation patches but are not mentioned in the corresponding documentation changes. This discrepancy poses a challenge for accurately evaluating LLMs: even if the generated code is functionally correct, differing entity names may lead to failed test cases and false negatives in evaluation. One possible solution is to simply discard such instances, as done in prior work [29] However, this would discard many valid feature-adding instances and compromise the representativeness of the benchmark. Instead, we supplement

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**

the input by including newly introduced entity names if they are explicitly referenced in test files. This increases the benchmark's representativeness while reducing evaluation bias caused by superficial mismatches.

To implement this method, we perform a structured static analysis to identify critical missing names. Specifically, 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 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. We refer to such entity names as *identifier hints*, which are stored as auxiliary metadata and can optionally be appended to the model input during evaluation.

In addition, we observed that some documentation changes include PR numbers, URLs, or references to issue trackers. These artifacts may guide models to recall memorized content or inspire models to access online resources, thus posing the risk of data leakage. To mitigate this risk, we implement automated scripts to detect and mask such information in the documentation changes.

Through these phases, we ultimately obtained 634 task instances. A final breakdown of these task instances across projects is presented in Table 1.

2.2 Task Formulation

For each task in NoCode-bench, the system under test is given a documentation change, a complete codebase, and optional identifier hints constructed in Phase 5. Its task is to edit the codebase to implement the new feature described in the documentation change. In practice, the model can produce standard Git patch files as the representation of the edits.

Table 2: 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.

2.3 NoCode-bench Verified

NoCode-bench collects a large number (634) of feature addition PRs from the wild, offering broad coverage and simulating real-world development scenarios. However, evaluating LLMs on 634 tasks is expensive and time-consuming. Furthermore, the specifications and the oracle (i.e., the test cases for our benchmark) of some real-world feature addition tasks are not well-specified, making them very challenging to tackle. To provide a lightweight and dependable alternative for debugging, fine-grained analysis, and evaluation under limited resources, we construct a manually validated subset, namely NoCode-bench Verified. We refer to the full set as NoCode-bench Full. Together, the two sets offer complementary strengths, balancing broad coverage with lightweight evaluation.

2.3.1 Construction Process. We aim for the verified subset to include at least 100 high-quality task instances spanning as many different projects as possible to ensure diversity. The construction process is based on random sampling of candidate tasks, followed by manual verification to assess task quality. Specifically, to carry out the annotations, we recruited five annotators, all of whom are screened for qualifications, including at least three years of experience with Python and a bachelor's or higher degree in a relevant field. Following the annotation guidelines of the recently released SWE-bench Verified [11], each annotator is asked to label samples based on **the Clarity of the Task** and **the Accuracy of the Evaluation**. 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.

2.3.2 Annotation Criteria. Our criteria used for manual annotation follow those of SWE-bench Verified, as follows: **Clarity of the Task.** Following SWE-bench Verified, we assess whether the task input (i.e., documentation changes in our work) clearly specifies the intended functionality. Annotators act as senior software engineers with full access to the codebase and related documentation. They judge whether the task is sufficiently well-defined to enable

Table 3: 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	67	220
1	97	89	–
2	57	69	–
3 (worst)	25	13	18

a meaningful implementation attempt and assign a clarity score accordingly. The scoring options range from 0 to 3, with lower scores indicating clearer and more precise documentation changes.
Accuracy of the Evaluation. 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. Annotators apply the test patch to the codebase and assess whether the added tests are appropriately scoped. Scoring is based on the criteria in Table 2, with lower scores indicating more accurate and targeted coverage.

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.

2.3.3 Annotation Results. 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 3, 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. Finally, we obtain 114 samples, which constitute NoCode-bench Verified.

3 EXPERIMENTAL SETUP

3.1 Research Questions

We evaluate the performance of state-of-the-art (SOTA) LLMs on no-code feature addition with NoCode-bench. Specifically, we aim to address the following research questions (RQs):

- RQ1: What are the characteristics of tasks in NoCode-bench compared to SWE-bench?
- RQ2: How do state-of-the-art LLM-based techniques perform on NoCode-bench Verified?
- RQ3: How do state-of-the-art LLM-based techniques perform on NoCode-bench Full?
- RQ4: What are the primary reasons for LLM failures on NoCode-bench?

3.2 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 very long contexts. Considering these requirements, we selected six state-of-the-art LLMs for evaluation. Three of them are open-source models, i.e., DeepSeek-v3-0324, DeepSeek-R1-0528, and Qwen3-325B-A22B, where DeepSeek-R1 is a reasoning model and Qwen3-325B-A22B is used in thinking mode. The other three are closed-source models, i.e., Claude-4-Sonnet, GPT-4o-2024-11-20, and Gemini-2.5-Pro-Exp-03-25, where Gemini-2.5-Pro is a reasoning model. For brevity, version numbers will be omitted when referencing model names throughout the paper. As these models provide a diverse spectrum in terms of size and training paradigms, we can explore how different architectures and capabilities impact the performance on no-code feature addition tasks.

3.3 Scaffolds

Due to the complexity of no-code feature addition tasks, directly prompting LLMs to generate the required changes is infeasible. Therefore, we adopt two state-of-the-art scaffolds for solving SE tasks: the pipeline-based method, **Agentless** [34], and the agent-based method, **OpenHands** [33]. They respectively represent two distinct paradigms to automate SE tasks using LLMs, and both have attained top scores on SWE-bench.

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. [14], 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.

3.4 Evaluation Metrics

Following prior studies [14, 16, 43], our end-to-end evaluation primarily considers the success rate of feature addition tasks (**Success%**), the success rate of patch application (**Applied%**), and the average token cost (**#Token**). In addition, we introduce three additional metrics to provide a more in-depth analysis of test case outcomes during evaluation.

- **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 F2P tests. Specifically, we calculate FV from both macro and micro perspectives.

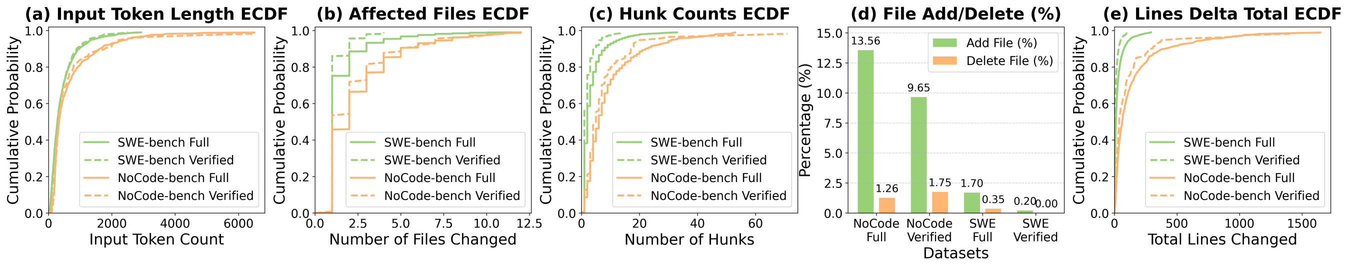


Figure 3: Comparison of golden patch distributions between NoCode-bench and SWE-bench along dimensions related to localization and editing difficulty.

– **Micro-level FV (FV-Micro):** The overall ratio of passed F2P tests to the total number of F2P tests across all instances: FV-Micro $= \frac{\sum_{i=1}^{N} \#\text{Passed}_i}{\sum_{i=1}^{N} \#\text{Total}_i}$, where #Passed$_i$ and #Total$_i$ denote the number of passed and total F2P tests for instance i, respectively.
– **Macro-level FV (FV-Macro):** The average F2P 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.

4 RESULTS

4.1 RQ1: Characteristics & Challenges

To better understand the unique characteristics and challenges posed by our benchmark, we analyze and compare 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. We regard the top 1% data as outliers and exclude them from Figure 3 to more clearly illustrate the differences.

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 additional challenges for the LLM's ability to comprehend the 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 more detailed and precise information, posing challenges to their abilities in long-text comprehension, identifying key information, and attending to fine-grained details.

Localization Difficulty. Accurate localization is a necessary condition for generating 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. This indicates 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. 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 need to add new files, while the percentage is only 1.70% in SWE-bench Full, which further increases localization complexity. These more diverse and structurally disruptive edits pose greater challenges for current models and frameworks.

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 relatively 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 require stronger code generation capabilities and greater coherence across multi-line edits.

RQ1 Summary: Compared to SWE-bench, NoCode-bench presents more complex challenges in understanding task input, localization, and code editing. The longer task input and higher frequency of multi-file changes, structurally diverse edits like file additions/deletions, and larger edit sizes collectively underscore the need for more advanced techniques to support real-world no-code feature addition tasks.

4.2 RQ2: Performance on NoCode-bench Verified

We evaluate six advanced LLMs on NoCode-bench Verified using two open-source scaffolds, and the results are shown in Table 4. We find that even the most advanced LLMs still exhibit limited performance in no-code feature addition. Specifically, the Claude-4-Sonnet model with Agentless scaffold achieves the highest success rate of 28.07% on NoCode-bench Verified, ranking first. Moreover, the Agentless scaffold achieves an average patch application rate of 98.69%, whereas OpenHands produces applicable patches for only 60.67% of the tasks on average. For regression tests, 75.15% of

Table 4: 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-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
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
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
Average	18.86%	98.69%	75.15%	8.37%	28.68%	50.68%	0.28M	9.50%	60.67%	55.26%	2.64%	14.45%	40.35%	0.77M

the patches generated by Agentless passed all regression tests on average, compared to only 55.26% for OpenHands. In terms of full F2P test suite pass rate, Agentless achieves 8.37% (FV-Micro), with an average of 28.68% (FV-Macro) of F2P tests passed per instance, while OpenHands achieves 2.64% and 14.45%, respectively. These results show that Agentless outperforms OpenHands on NoCode-bench Verified. 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.

Notably, the Gemini-2.5-Pro model with the OpenHands scaffold failed to solve any tasks on NoCode-bench Verified. Through manual inspection, we find that Gemini-2.5-Pro is almost entirely unable to generate correct invocation formats for any of the tools used by OpenHands. This directly prevents it from accessing and editing the repository. We further discuss this issue in Section 4.4.

We further compare the performance of these LLMs on SWE-bench Verified with their performance on NoCode-bench Verified. For example, OpenHands+Claude-4-Sonnet achieves a 70.4% success rate on SWE-bench Verified, but only 25.44% on NoCode-bench Verified. This indicates that, compared to issue-solving, no-code feature addition tasks represent a distinct and unsolved challenge, offering a new perspective for evaluating the SE capabilities of LLMs.

RQ2 Summary: The state-of-the-art LLMs perform poorly on feature addition tasks in no-code development scenarios. This demonstrates that NoCode-bench introduces a novel and challenging task that enables the evaluation of LLMs' SE capabilities from a unique perspective.

4.3 RQ3: Performance on NoCode-bench Full

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 (about USD 1,190) and Gemini-2.5-Pro (about USD 600), we leave their evaluation on NoCode-bench Full for future work. Considering the better performance of Agentless, we use it as the scaffold to evaluate the remaining four models on NoCode-bench Full, and report the results in Table 5.

Table 5: 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
GPT-4o	9.31%	97.00%	71.14%	8.68%	17.81%	43.58%	0.11M
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
Average	10.65%	98.03%	75.79%	11.50%	20.86%	43.85%	0.30M

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 accuracy of modified files drops from 49.49% to 43.85%. We further investigate the causes of this phenomenon. 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, because NoCode-bench Full is unverified, it contains some noise, including unclear documentation changes or low-coverage regression tests, which make it more representative of real-world, no-code feature addition tasks, but harder for LLMs to solve. In contrast, NoCode-bench Verified is a high-quality and more reliable subset, enabling faster and more controlled evaluation of LLMs' ability to solve no-code feature addition tasks.

We also find that the four models vary in robustness when handling real-world feature addition tasks in no-code scenarios. 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 on NoCode-bench Verified, while GPT-4o outperforms Qwen3-235B on NoCode-bench Full. These results demonstrate that GPT-4o performs more robustly in application scenarios that are closer to real-world conditions compared to DeepSeek-v3 and Qwen3-235B.

RQ3 Summary: Since NoCode-bench Full is a large-scale dataset containing more complex edits and more diverse task scenarios, state-of-the-art LLMs perform worse on NoCode-bench Full compared to NoCode-bench Verified. Moreover, NoCode-bench Full is closer to real-world, no-code feature

Table 6: Performance comparison 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	
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)
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)
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)
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)
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-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)
NoCode-bench Full					
Total Instances		290		344	
DeepSeek-R1	Agentless	24.83% (72)	96.55% (280)	6.40% (22)	95.35% (328)
DeepSeek-v3	Agentless	20.00% (58)	100.00% (290)	4.94% (17)	99.71% (343)
GPT-4o	Agentless	15.17% (44)	98.97% (287)	4.36% (15)	95.35% (328)
Qwen3-235B	Agentless	11.38% (33)	99.66% (289)	2.62% (9)	99.13% (341)

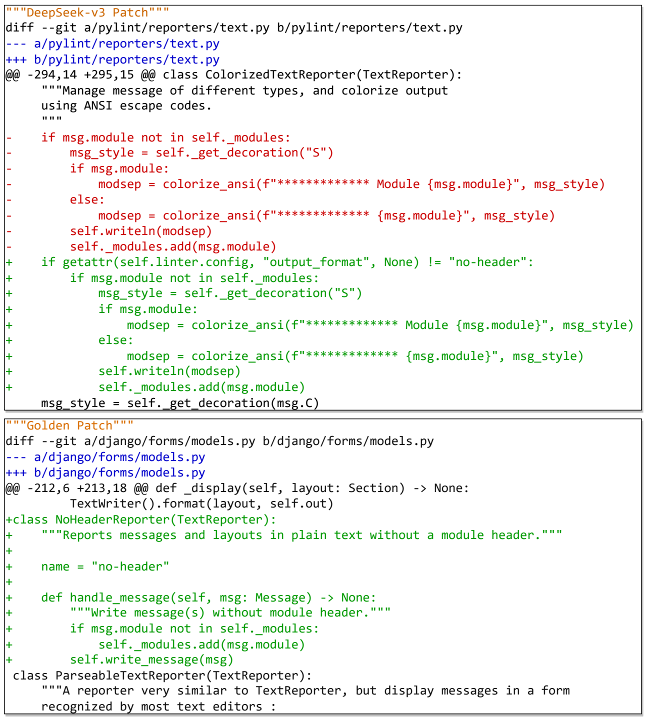
addition scenarios, enabling a more comprehensive evaluation of LLMs’ feature addition capabilities.

4.4 RQ4: Failure Analysis

To investigate why LLMs fail to solve tasks on NoCode-bench, we manually examine the failed instances and identify the following primary reasons for LLM failures on NoCode-bench.

Lack of cross-file editing capability. In NoCode-bench Verified, only about half of the instances (61/114) can be solved by editing a single file; in NoCode-bench Full, only 45.7% of the instances involve single-file edits. Therefore, 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” (number of modified files = 1) and “multi-file” (number of modified files > 1). We analyze the performance of LLMs on “single-file” and “multi-file” tasks, and present the results in Table 6. We find that the best-performing model on NoCode-bench Verified achieves only a 9.43% success rate on “multi-file” tasks. On NoCode-bench Full, it solves only 6.40% of instances involving “multi-file”. Moreover, model performance is significantly worse in terms of success rate in scenarios that require multi-file edits. These results indicate that current LLMs tend to favor single-file edits, while NoCode-bench is a highly challenging task that relies heavily on cross-file retrieval and multi-file editing. Therefore, the lack of cross-file editing capability is one of the main reasons for LLM failures on NoCode-bench.

Lack of comprehensive understanding of the codebase structure. The code changes to add a new feature are never expected to compromise the original functionality of the software, which poses a significant challenge to the LLM’s architectural understanding. However, through manual inspection, we find that LLMs often attempt to implement new features by directly modifying existing functionality. A typical example is the patch generated by DeepSeek-v3+Agentless on the task “pylint-7869”. Figure 4 highlights a critical difference between the patch generated by DeepSeek and the golden patch. This task requires the addition of a new “no-header” option, which hides the module name header information in Pylint’s output. 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 compromises the original functionality. Although it passes all F2P tests related to the new feature, it results in numerous regression test failures. This case shows that NoCode-bench challenges LLMs to understand the architecture of the entire codebase. However, LLMs often lack this capability and generate edits that compromise existing functionality. Therefore, the lack of architectural understanding is a key reason for LLM failures on NoCode-bench.



```
"""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 4: A failure example of *pylint-dev__pylint-7869* caused by the LLM’s lack of architectural understanding.

Insufficient tool-calling capability. This type of error occurs during the tool-calling phase of the OpenHands CodeAct Agent. 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 format. This makes the system highly dependent on the LLM’s output formatting capability. However, nearly all LLMs fail to invoke tools in the correct format, with Gemini-2.5-Pro being particularly prone to this issue, which directly results in a success rate of 0 on NoCode-bench Verified. For example, when OpenHands attempts to solve the task *astropy-9425*, the agent is expected to autonomously perform tool calls based on the task description to complete a feature addition task. 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 case shows that the model's tool-calling capability greatly affects the agent's ability to operate on the codebase and can even directly lead to task failure.

RQ4 Summary: Feature addition tasks rely on the LLM's ability to perform cross-file modifications and its comprehensive understanding of project architecture. State-of-the-art LLMs perform poorly due to the lack of both capabilities. In addition, the difficulty LLMs face in generating correct tool-calling formats to drive agents also serves as a performance bottleneck.

5 THREATS TO VALIDITY

Internal Validity. *Task Quality:* In practice, the documentation change in a PR may be incomplete or not clear enough to implement the feature implemented in this PR. Also, the test suite of a project may not fully test a feature. To mitigate this threat, we curate a manually validated subset, NoCode-bench Verified, to increase the reliability of the evaluation. *Data Leakage:* The instances in NoCode-bench are collected from historical commits in specific GitHub repositories. As GitHub data is widely used to train state-of-the-art LLMs, there is a potential risk of data leakage. Existing benchmarks that are constructed from GitHub data all face this threat. To help mitigate this risk, we mask PR numbers during dataset construction. The low success rates of the best-performing LLMs (29% or less) suggest the impact of data leakage to be limited.
External Validity. *Generalization:* Our benchmark currently focuses on well-maintained open-source Python projects that provide structured release notes. While this design ensures high-quality and interpretable data, it inevitably restricts project diversity and language coverage. However, Python is one of the most popular programming languages, and the construction methodology of NoCode-bench is language-agnostic and can be adapted to other ecosystems in future work. *Used Scaffolds:* We evaluate the performance of the selected LLMs using only the Agentless and OpenHands scaffolds. Their performance on other scaffolds remains unknown. However, Agentless and OpenHands represent the state-of-the-art in their respective categories. Their limited performance across different LLMs indicate that NoCode-bench poses new challenges for LLM-based software engineering methods.

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 [16, 23] 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 [40, 43, 44] and multimodal [14, 28, 39] scenarios, promoting broader language and scenario coverage in issue-solving tasks. GITS-Eval [29] proposes a similar benchmark constructed within an industrial setting, which also focuses on issue solving. However, in practice, about 60% of maintenance effort involves feature enhancement rather than defect correction [13, 26], which motivates the need for dedicated benchmarks targeting feature addition tasks. FEA-Bench [19] tries to address this gap by evaluating LLMs on feature addition tasks that come with detailed code-level specifications, including function/class signatures, docstrings, as well as the title and description of the PR that adds a feature.

Table 7: Comparison with existing benchmarks.

Aspect	SWE-bench	GITS-Eval	FEA-Bench	NoCode-bench
Development scenario	Traditional	Traditional	Traditional	No-code
Task type	Issue solving	Issue solving	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

To highlight the distinctive features that set NoCode-bench apart from SWE-bench, FEA-Bench, and GITS-Eval, we present a detailed comparison in Table 7. Compared to FEA-Bench, NoCode-bench adopts a more realistic no-code setting where only user-facing documentation changes are provided as input, rather than relying on predefined code-level specifications. For instance selection, NoCode-bench does not constrain instances to those introducing new components, recognizing that many feature additions involve extending or modifying existing code without adding entirely new functions or classes. In addition, NoCode-bench identifies feature-adding PRs based on the labels in developer-maintained release notes, which are reliable. In contrast, FEA-Bench relies on LLM-based classification, which may introduce noise [37, 45] and involve non-feature-adding instances. Finally, we manually construct a high-quality, verified subset to enable lightweight yet reliable evaluation. We open-source the entire construction pipeline and evaluation code of NoCode-bench to facilitate the progress in no-code feature addition.

6.2 LLM-based Methods for SE

Recently, various LLM-based methods [5, 17, 30, 34–36, 38, 46] have been proposed to address SE tasks, primarily including pipeline-based methods and agent-based methods. For *pipeline-based methods*, researchers typically decompose the task into stages, such as localization, repair, and validation based on prior knowledge, and require the LLM to execute each stage in a fixed workflow. Examples include Agentless [34], RepoGraph [24], and PatchPilot [18]. For *agent-based methods*, researchers equip agents with a variety of tools (e.g., bash, edit, language server) to enable access to and modification of the codebase. These approaches rely on the LLM's autonomous decision-making and tool invocation to generate patches. Examples include agents for bug repair and issue solving, e.g., SWE-Agent [38], RepairAgent [7], OpenHands [33], and TRAE [9], and for other software engineering tasks, e.g., ExecutionAgent [8]. Although these methods are originally designed with a traditional coding scenario in mind, their underlying mechanisms are also applicable to no-code development scenarios, which allows them to be adapted to NoCode-bench. However, since feature addition tasks in NoCode-bench require a more comprehensive understanding of the codebase and stronger cross-file editing capabilities, NoCode-bench presents new and unique challenges.

7 CONCLUSION

We introduce NoCode-bench, a large-scale and challenging benchmark for evaluating LLMs' software engineering capabilities in no-code feature addition. NoCode-bench is constructed with a systematic five-phase pipeline, including project selection, instance collection, environment construction, instance filtering, and input refinement. 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 an evaluated system to generate a patch that adds a new feature. We evaluate six state-of-the-art LLMs using two representative scaffolds, Agentless and OpenHands. Our results show that the best-performing model succeeds only 28.07% of instances in NoCode-bench Verified, and 14.83% in NoCode-bench Full. This demonstrates that no-code feature addition is a uniquely difficult and unsolved problem. Moreover, we analyze the failure cases of LLMs on NoCode-bench, aiming to guide future improvements in cross-file editing, codebase structure understanding, and tool-calling capabilities.

DATA AVAILABILITY

Our code and data are available at https://github.com/NoCode-bench/NoCode-bench.

REFERENCES

[1] 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)*. 1199–1210.

[2] Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avi Shinnar, Saurabh Sinha, and Martin Hirzel. 2025. Otter: Generating Tests from Issues to Validate SWE Patches. In *International Conference on Machine Learning*.

[3] Anaconda Inc. 2025. Anaconda Documentation. https://www.anaconda.com/.

[4] Anthropic. 2024. The Claude 3 Model Family: Opus, Sonnet, Haiku. https://www.anthropic.com/news/claude-3-family.

[5] Aorwall. 2025. Moatless Tools. https://github.com/aorwall/moatless-tools.

[6] 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 (2020), 1834–1852.

[7] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *International Conference on Software Engineering (ICSE)*.

[8] Islem Bouzenia and Michael Pradel. 2025. You Name It, I Run It: An LLM Agent to Execute Tests of Arbitrary Projects. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA047, 23 pages.

[9] Trae by ByteDance. 2025. Trae: Collaborate with Intelligence. Official Website. https://www.trae.ai/

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[11] 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. https://openai.com/index/introducing-swe-bench-verified/

[12] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*. 345–356.

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

[14] Lianghong Guo, Wei Tao, Runhan Jiang, Yanlin Wang, Jiachi Chen, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. 2025. OmniGIRL: A Multilingual and Multimodal Benchmark for GitHub Issue Resolution. In *Proceedings of the 34rd ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[15] Martin Hirzel. 2023. Low-code programming models. *Commun. ACM* 66, 10 (2023), 76–85.

[16] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[17] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[18] Hongwei Li, Yuheng Tang, Shiqi Wang, and Wenbo Guo. 2025. PatchPilot: A Cost-Efficient Software Engineering Agent with Early Attempts on Formal Verification. In *Forty-second International Conference on Machine Learning*.

[19] Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. 2025. FEA-Bench: A Benchmark for Evaluating Repository-Level Code Generation for Feature Implementation. arXiv:2503.06680

[20] Niels Mündler, Mark Müller, Jingxuan He, and Martin Vechev. 2024. SWT-bench: Testing and validating real-world bug-fixes with code agents. *Advances in Neural Information Processing Systems* 37 (2024), 81857–81887.

[21] Noor Nashid, Islem Bouzenia, Michael Pradel, and Ali Mesbah. 2025. Issue2Test: Generating Reproducing Test Cases from Issue Reports. *arXiv preprint arXiv:2503.16320* (2025).

[22] OpenAI. 2024. GPT-4o System Card. arXiv:2410.21276 [cs.CL] https://arxiv.org/abs/2410.21276

[23] OpenAI. 2024. SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/.

[24] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph. In *Proceedings of The Thirteenth International Conference on Learning Representations*.

[25] Pallets. 2025. Flask: Micro Framework for Building Web Applications. https://flask.palletsprojects.com/en/stable/.

[26] 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)*. 226–229.

[27] 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*. 837–852.

[28] Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. 2025. SWE-PolyBench: A multi-language benchmark for repository level evaluation of coding agents. arXiv:2504.08703

[29] 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. *arXiv preprint arXiv:2501.07531* (2025).

[30] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2024. Specrover: Code intent extraction via llms. *arXiv preprint arXiv:2408.02232* (2024).

[31] 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)*. 171–178.

[32] SymPy Development Team. 2025. SymPy: Python Library for Symbolic <athematics. https://docs.sympy.org/latest/index.html.

[33] 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. 2025. OpenHands: An Open Platform for AI Software

Developers as Generalist Agents. In *The Thirteenth International Conference on Learning Representations*.

[34] 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 (2025), 801–824.

[35] Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. 2025. SWE-Fixer: Training Open-Source LLMs for Effective and Efficient GitHub Issue Resolution. *arXiv preprint arXiv:2501.05040* (2025).

[36] Chuyang Xu, Zhongxin Liu, Xiaoxue Ren, Gehao Zhang, Ming Liang, and David Lo. 2025. FlexFL: Flexible and Effective Fault Localization With Open-Source Large Language Models. *IEEE Transactions on Software Engineering* 51, 5 (2025), 1455–1471.

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

[38] John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[39] 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. 2025. SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains?. In *The Thirteenth International Conference on Learning Representations*.

[40] John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. SWE-smith: Scaling Data for Software Engineering Agents. arXiv:2504.21798

[41] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

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

[43] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. 2025. Multi-SWE-bench: A Multilingual Benchmark for Issue Resolving. arXiv:2504.02605

[44] Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, et al. 2024. Swe-bench-java: A github issue resolving benchmark for java. *arXiv preprint arXiv:2408.14354* (2024).

[45] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023. Siren's song in the AI ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219* (2023).

[46] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.