Doc2Feat-Bench: Evaluating Documentation-Driven Feature Addition

Zhonghao Jiang
zhonghao.j@zju.edu.cn
College of Computer Science and Technology and the State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, China

Le Deng
dengle@zju.edu.cn
College of Computer Science and Technology and the State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, China

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

Michael Pradel
michael@binaervarianz.de
CISPA Helmholtz Center for Information Security
Stuttgart, Germany

Zhongxin Liu*
liu_zx@zju.edu.cn
College of Computer Science and Technology and the State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, China

Abstract

Documentation changes in mature software projects often describe newly introduced or modified behavior. This makes them a natural basis for *documentation-driven feature addition*, where software engineering agents implement features from public-facing documentation updates. However, existing benchmarks for agentic software development primarily rely on issue reports written for maintainer coordination and often contain implementation-level details. Motivated by this observation, this work introduces Doc2Feat-Bench, a benchmark for documentation-driven feature addition. Doc2Feat-Bench consists of 634 tasks across 10 mature open-source software projects, involving about 114k code changes in total. Each task pairs a documentation change with the corresponding implementation and developer-written tests, enabling evaluation of whether a software engineering agent can implement the feature described by the documentation diff. The benchmark is constructed through a five-phase pipeline, starting from release notes, and offers broad coverage across libraries, developer tools, and frameworks while preserving real-world development settings. To facilitate lightweight and reliable evaluation under limited resources, we further curate a human-validated subset named Doc2Feat-Bench Verified. It contains 114 high-quality instances whose task clarity and evaluation validity are manually verified. We use Doc2Feat-Bench to assess a range of state-of-the-art software engineering agents with two widely used scaffolds. Experimental results show that despite significant token consumption, the best task success rate remains as low as 37.72% (OpenHands with Qwen3-Coder-480B). Our analysis reveals that software engineering agents face key challenges in performing cross-file edits, understanding existing code modules, and correctly interpreting documentation semantics.

CCS Concepts

• **Software and its engineering** → **Automatic programming**.

Keywords

Agents, Documentation-Driven, Feature Addition

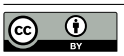
ACM Reference Format:
Zhonghao Jiang, Le Deng, Jialun Cao, Michael Pradel, and Zhongxin Liu. 2026. Doc2Feat-Bench: Evaluating Documentation-Driven Feature Addition. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837489

1 Introduction

Feature addition refers to the activity of introducing new functionality or modifying existing behavior in software systems to meet evolving user needs [20, 39]. It is a central activity in software evolution, accounting for a large portion of software maintenance efforts [17, 35]. Traditionally, feature addition is manually performed by developers. *But what if new features could be added to a software project simply by updating its public documentation?* In this setting, documentation updates act as executable, public-facing feature specifications that software engineering agents translate into code changes. As software engineering agents are increasingly applied to repository-level development tasks, evaluating their ability to implement new features accurately becomes important. We refer to the setting in which agents implement features from documentation updates as *documentation-driven feature addition*. To support meaningful evaluation in this setting, high-quality benchmarks are essential, as they help assess agent capabilities, identify limitations, and guide future improvements.

Existing benchmarks for agentic software development primarily rely on issue reports as task inputs [21]. However, issue reports

*Corresponding Author.



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837489

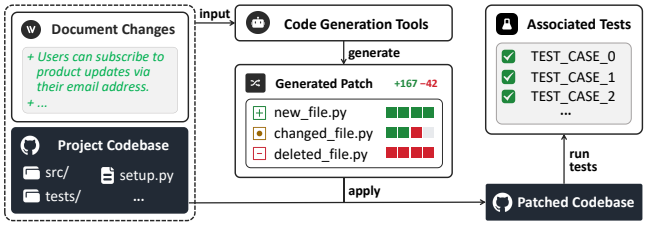


Figure 1: Overview of documentation-driven feature addition tasks.

are typically maintainer-facing. They are designed for coordination among developers [15], are typically authored by individuals familiar with the codebase [10], and often include implementation-level technical details such as reproduction steps, stack traces, and test cases [13, 38]. In addition, issue-based benchmarks are focused primarily on bug fixing or issue resolution, with insufficient attention to feature addition. Therefore, existing benchmarks do not fully capture the needs of evaluating documentation-driven feature addition.

The key to evaluating documentation-driven feature addition lies in identifying task inputs that reflect externally visible feature requirements. Documentation provides a natural and effective source of such inputs. It is written for external users and integrators of software projects [1, 5] and serves as an authoritative description of software functionality and expected behavior. Emerging development practices, such as README-driven development [33], OpenAPI-based workflows [31], and spec-driven development [16], have followed a documentation-first manner. As software evolves, documentation is often updated to reflect newly introduced or modified features [7]. These documentation changes specify intended behavioral changes in a public-facing form, making them a valuable source for constructing feature addition tasks.

Based on this observation, we introduce Doc2Feat-Bench, a benchmark for documentation-driven feature addition. Doc2Feat-Bench identifies feature addition tasks through release notes and constructs task inputs from *changes in public-facing software documentation*, which serve as authoritative specifications of software functionality [2]. It consists of 634 carefully collected tasks derived from 10 mature open-source software projects hosted on GitHub, spanning libraries, developer tools, and frameworks. As illustrated in Figure 1, each instance takes documentation changes as input and expects a software engineering agent 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 Doc2Feat-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 Doc2Feat-Bench Verified and Doc2Feat-Bench Full to systematically evaluate a range of state-of-the-art software engineering agents with the Agentless [43] and OpenHands [42] scaffolds. Evaluation results show that, despite incurring high token consumption, the best success rate is only 37.72%, achieved when using Qwen3-Coder-480B with the OpenHands scaffold. In comparison, when evaluated on SWE-bench, a benchmark collected from similar projects, the same model and scaffold achieve a success rate of 69.60%. This stark performance gap highlights documentation-driven feature addition as a substantially more challenging and complementary benchmark setting. Our analysis further reveals that software engineering agents face key challenges in performing cross-file edits, understanding existing code modules, and correctly interpreting documentation semantics.

In summary, our main contributions are:

- We introduce Doc2Feat-Bench, a documentation-driven benchmark for feature addition in mature open-source software projects, using documentation diffs as public-facing feature specifications.
- 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 software engineering agents on Doc2Feat-Bench, analyze their performance from both quantitative and qualitative perspectives, and identify key factors that affect agent performance.

2 Doc2Feat-Bench

Doc2Feat-Bench focuses on evaluating software engineering agents in documentation-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. This section outlines the construction process of Doc2Feat-Bench and its task formulation. In addition, we describe how we constructed a reliable subset, Doc2Feat-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 Doc2Feat-Bench through the five phases shown in Figure 2.

2.1.1 Phase 1: Project Selection. The project selection begins with the 12 projects used in SWE-bench [23], which are actively 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. Well-maintained release notes often annotate each change with its type (e.g., "feature", "enhancement", "bug fix", etc.), 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 documentation-driven feature addition, we apply additional criteria when selecting projects. Specifically, we require that projects satisfy the following three conditions:

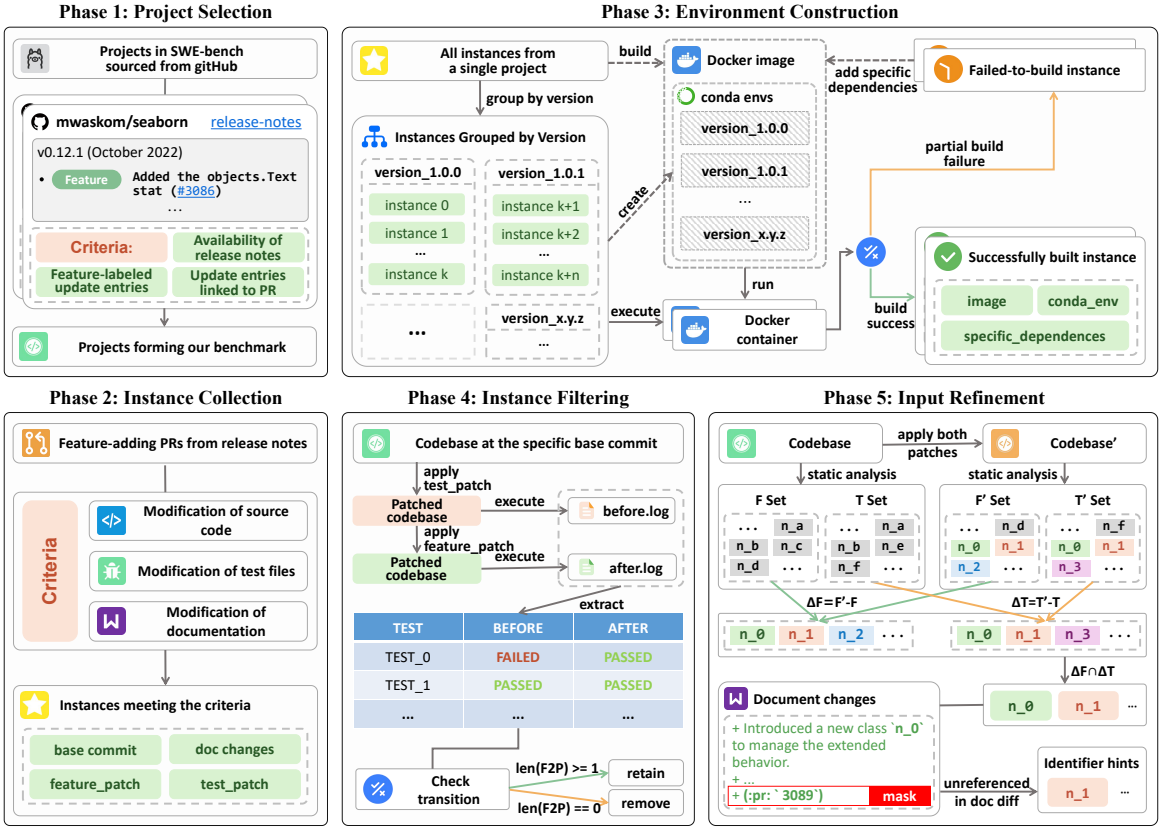


Figure 2: The workflow of building our benchmark.

- **Availability of release notes:** Prior work [9, 49] 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 externally visible 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. Without such labels, it is often unclear whether a code change corresponds to a feature addition or a bug fix.
- **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 changelog 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 all the above conditions. The `flask` [32] and `sympy` [40] projects lack feature-labeled annotations and are thus excluded.

2.1.2 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. Then, we apply the following three 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 considers only source code and test modifications, we additionally require documentation changes, which serve as the task specification provided as input. Compared to issue reports, documentation changes more directly describe intended feature behavior without exposing implementation details. Additionally, we set a cutoff date of August 15, 2024, 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. 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 1,571 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 [47]. 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 [36].

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.

Table 1: 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 are about.

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. Among the 634 instances in Doc2Feat-Bench, 276 instances show such errors in before.log.

2.1.5 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 [23], would remove legitimate feature additions and thus reduce benchmark coverage.

To preserve these instances while avoiding superficial evaluation failures, 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 validation 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. Accordingly, we provide two benchmark variants: one with *identifier hints* and one without. This design allows users to choose whether to include such information, depending on their tolerance for potential data leakage.

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.

Together with previous phases, this yields the final set of 634 documentation-driven feature addition tasks used in Doc2Feat-Bench. A final breakdown of these task instances across projects is presented in Table 8 (see Appendix A).

2.2 Doc2Feat-Bench Verified

Doc2Feat-Bench collects a large number (634) of feature addition PRs from the wild, offering broad coverage and simulating real-world development scenarios. However, evaluating software engineering agents 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 Doc2Feat-Bench Verified. We refer to the full set as Doc2Feat-Bench Full. Together, the two sets offer complementary strengths, balancing broad coverage with lightweight evaluation.

2.2.1 Construction Process. To ensure that the verified subset is representative of the full set, it should include at least 84 instances, which is the minimum sample size required for 634 Doc2Feat-Bench Full instances at a 95% confidence level with a 10% error margin, following common practice in empirical software engineering [3, 8]. We perform random sampling, followed by manual verification to assess task quality. To ensure project diversity, we draw a stratified random sample by project from Doc2Feat-Bench Full. Considering that low-quality samples may be filtered out during manual verification, we take up to 35 task instances per project, yielding a total of 238 candidate tasks.

Specifically, to carry out the annotations, we recruited five annotators, all of whom were screened for qualifications, including at least three years of experience with Python and a bachelor's degree or higher in a relevant field. Following the annotation guidelines of the recently released SWE-bench Verified [29], each annotator is asked to label samples based on **the Clarity of the Task** and **the Accuracy of the Evaluation**. 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.

Table 2: 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

2.2.2 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 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 1, 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 reason why 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.

2.2.3 Annotation Results. After all the samples are 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 2, a total of 106 instances were removed due to unclear task descriptions or inaccurate evaluations, and 18 instances were

Table 3: Common quality issues in filtered instances.

Type	Share	Description
Task Clarity		
Changelog-only docs	42.9%	Brief changelog-only documentation without usage guidance, parameter semantics, or examples.
Non-feature changes	19.5%	Internal refactoring, performance optimization, or deprecation changes without user-visible functionality.
Scope mismatch	15.6%	Implementation scope substantially exceeds the behavior specified by the documentation diff.
Evaluation Accuracy		
Limited tests	30.1%	Minimal tests covering only basic functionality, without boundary conditions (e.g., empty or extreme inputs, alternative parameter settings) or error cases.
Adaptation-only tests	21.9%	Test changes that mainly adapt identifiers or imports without validating new behavior.
Missing integration tests	16.4%	Unit tests for a new module without integration coverage for modified existing files.

excluded due to other major problems. The removed 106 instances consist of 77 instances flagged on task clarity and 73 on test accuracy, which overlap in 44 cases. Among the 18 excluded for other major problems, 3 were flagged unanimously by both annotators and thus left unscored along the two dimensions, so the Task Clarity and Test Accuracy columns in Table 2 total 235 rather than 238. Finally, we obtain 114 samples, which constitute Doc2Feat-Bench Verified. During verification, Cohen's kappa is 0.4193, indicating moderate agreement and closely matching prior work (0.3906 on SWE-bench Verified [29]), which supports the reliability of our process. To further validate the reliability of the evaluation, we measured the line-level coverage of each golden feature patch achieved by its F2P tests. On Doc2Feat-Bench Verified, the median and mean coverage reach 94.60% and 84.90%, respectively, comparable to prior work (96.94% and 72.95% on SWE-bench Verified). This indicates that the retained F2P tests provide reliable evaluation signals.

2.2.4 Analysis of Filtered Instances. To better understand why instances were filtered out during the construction of Doc2Feat-Bench Verified, two authors manually analyzed the prior annotation results, including annotator scores and explanations, together with each instance's documentation change, golden patch and test patch. This analysis covered the 77 instances filtered out under the task clarity dimension and the 73 filtered out under the evaluation accuracy dimension. For each of the two dimensions, the two authors first jointly inspected 30 pilot instances to derive an initial coding scheme. As the pilot annotation reveals a long-tail distribution, where the top-3 issue types cover 77% and 71% of the pilot instances along the two dimensions while each remaining type covers only 1.4 and 1.8 instances on average, we focus the analysis on the top-3 types in each dimension (shown in Table 3). Specifically, using this scheme, both authors independently categorized all filtered instances along the respective dimensions, reaching high agreement, with Cohen's kappa of 0.932 for task clarity and 0.913 for evaluation accuracy. All disagreements were resolved through discussion.

The upper part of Figure 3 illustrates how task clarity issues arise from the gap between the documentation snippet and the implementation requirement. In the *changelog-only docs* example, the documentation only states that `Birch` supports `n_clusters=None`, but the implementation requires knowing that this setting should

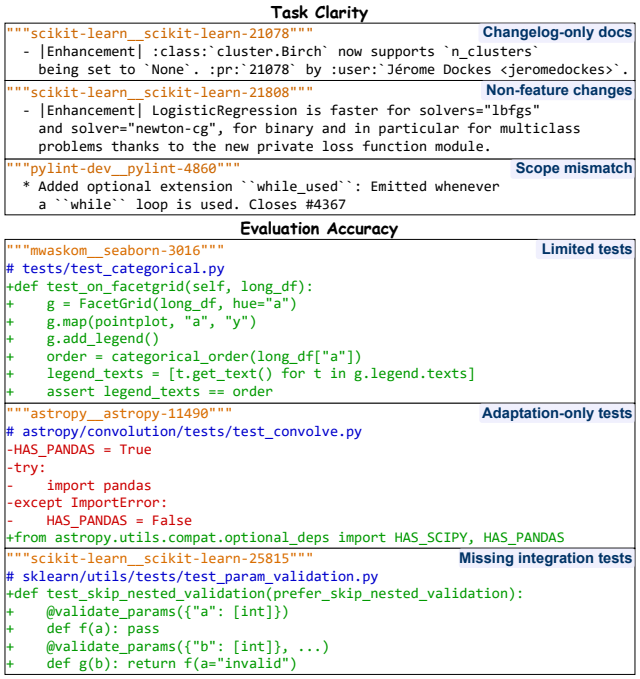


Figure 3: Representative examples of task-clarity and evaluation-accuracy issues in filtered instances.

bypass global clustering and directly form flat clusters from `_CFSubcluster` objects. In the *non-feature changes* example, the documentation says that `LogisticRegression` is faster due to a new private loss-function module. The actual change is an internal optimization with no new user-facing API or behavior to implement. In the *scope mismatch* example, the documentation presents the change as adding a `while_used` checker, while the PR also rewrites broader `while`-loop patterns across the codebase, so the documentation substantially understates the implementation scope.

The lower part of Figure 3 shows how evaluation accuracy issues arise when tests fail to exercise the intended behavior. In the *limited tests* example, the added test checks only one `FacetGrid` legend-order case for `pointplot`, leaving other label settings, hue interactions, and error cases uncovered. In the *adaptation-only tests* example, the patch repeatedly replaces local dependency checks such as `HAS_PANDAS` with imports from `optional_deps`. These changes adapt existing tests to a new import path but never validate the lazy access-time dependency checks introduced by the code patch. In the *missing integration tests* example, the tests use toy functions to exercise nested parameter validation in isolation, while the real integration points, such as estimator `fit()` methods and `non_negative_factorization`, remain untested.

3 Benchmark Characteristics

To better understand unique characteristics and challenges posed by Doc2Feat-Bench, we analyze the distribution of instances in Doc2Feat-Bench and SWE-bench across dimensions related to input complexity, localization difficulty, and editing difficulty, as shown in Figure 4. For visualization purposes only, we omit extreme values

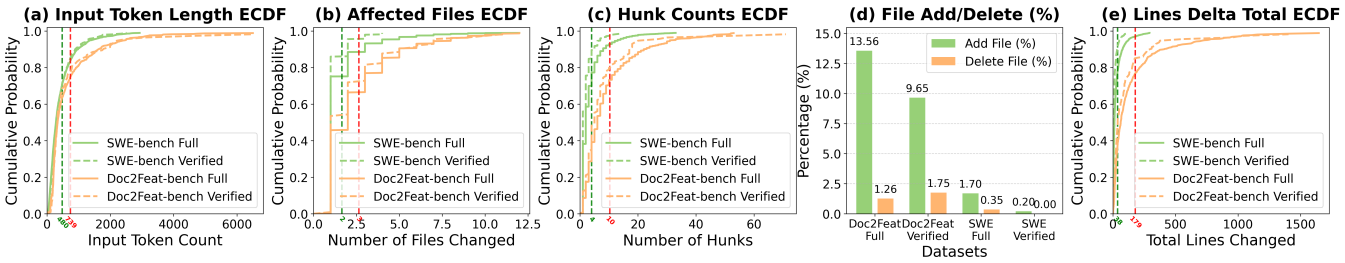


Figure 4: Comparison of golden patch distributions between Doc2Feat-Bench and SWE-bench along dimensions related to localization and editing difficulty. ECDF stands for "empirical cumulative distribution function".

(the top 1% of values in each dimension) from Figure 4 to prevent distortion; these values are included in all statistical analyses.

Input Complexity. Unlike SWE-bench, which uses issue descriptions as input, Doc2Feat-Bench takes documentation changes as input and requires software engineering agents to understand the semantic differences before and after the change. Long documentation changes pose extra challenges for agents to comprehend the task and generate appropriate outputs. As shown in Figure 4(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 documentation changes in Doc2Feat-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 Doc2Feat-Bench require agents to capture information from longer and more complex task inputs, posing greater challenges to 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 4(b,c), on average, golden patches in Doc2Feat-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 Doc2Feat-Bench and 1.24 for SWE-bench, with corresponding hunk counts of 8.96 and 2.43. Moreover, Figure 4(d) reveals that Doc2Feat-Bench includes a much higher proportion of patches that involve adding or deleting files. 13.56% of instances in Doc2Feat-Bench Full require new files, while the percentage is only 1.70% in SWE-bench Full. These results indicate that our benchmark requires agents 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 4(e). In Doc2Feat-Bench Full, golden patches involve an average of 179.12 lines of edits, and 179.22 lines in Doc2Feat-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 nearly all SWE-bench patches are under 200 lines, while almost 20% of Doc2Feat-Bench patches exceed that length. These larger modifications may require stronger code generation capabilities and greater coherence across multi-line edits.

4 Experimental Setup

4.1 Research Questions

- RQ1: How do state-of-the-art software engineering agents perform on Doc2Feat-Bench Verified?
- RQ2: How do state-of-the-art software engineering agents perform on Doc2Feat-Bench Full?
- RQ3: What are the reasons for agent failures on Doc2Feat-Bench?

4.2 Model Selection

Completing a task in Doc2Feat-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 models for evaluation on Doc2Feat-Bench Verified. Four of them are open-source models, i.e., DeepSeek-v3-0324 [26], DeepSeek-R1-0528 [18], Qwen3-235B-A22B [45], and Qwen3-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 [6], GPT-4o-2024-11-20 [28], Gemini-2.5-Pro [14], and GPT-5-mini [30], 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. Doc2Feat-Bench Full is significantly larger than Doc2Feat-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 USD 5,124) running on Doc2Feat-Bench Full, we focus on Doc2Feat-Bench Verified for those models.

4.3 Scaffolds

Due to the complexity of documentation-driven feature addition tasks, directly prompting models 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 [43], and an agent-based method, OpenHands [42]. They respectively represent two distinct paradigms to automate SE tasks with software engineering agents, 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

Table 4: Performance comparison across different software engineering agents on Doc2Feat-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

require modification, and then generates patches based on the localized content with the selected model. Following Guo et al. [19] and Zan et al. [50], we use Agentless-1.0 for evaluation.

- **OpenHands** is a widely adopted platform for building software development agents. We adapt the documentation changes in Doc2Feat-Bench as inputs to the OpenHands CodeAct Agent and utilize the standard evaluation Docker images provided by Doc2Feat-Bench as a sandbox environment to ensure the correct execution of tasks.

4.4 Evaluation Metrics

Following prior studies [19, 23, 50], our end-to-end evaluation primarily considers the success rate (i.e., the proportion of instances for which the given patch passes the accompanying 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 the completeness of an agent's implementation of a specific feature, which we measure based on the pass rate of F⟶P tests. It is reported at both micro and macro levels to reflect overall test-level performance and instance-level robustness, respectively.

- **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. N is the total number of instances.
- **Macro-level FV (FV-Macro):** The average F⟶P pass rate across instances: FV-Macro $= \frac{1}{N} \sum_{i=1}^{N} \frac{\#\text{Passed}_i}{\#\text{Total}_i}$.

5 Results

5.1 RQ1: Performance on Doc2Feat-Bench Verified

The evaluation results are shown in Table 4. We find that even the most advanced software engineering agents still exhibit limited performance in documentation-driven feature addition. Specifically, among the evaluated models, Qwen3-Coder-480B achieves the highest success rate of 37.72% on Doc2Feat-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 Doc2Feat-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 Doc2Feat-Bench Verified, implying that documentation-driven feature addition tasks present a substantially greater challenge compared to issue resolution tasks.

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 Doc2Feat-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 Doc2Feat-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, Gemini-2.5-Pro with OpenHands solved no tasks on Doc2Feat-Bench, largely because it failed to produce valid tool invocation formats, preventing repository access and modification. We further discuss this issue in Appendix B.

> **RQ1 Summary:** State-of-the-art software engineering agents perform poorly on documentation-driven feature addition tasks. Even the best-performing model, Qwen3-Coder-480B with OpenHands, achieves only 37.72% success rate on

Table 5: Performance comparison across different software engineering agents on Doc2Feat-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

Doc2Feat-Bench Verified, while GPT-4o and Gemini-2.5-Pro achieve 5.26% and 0%, respectively. This demonstrates that Doc2Feat-Bench introduces a novel and challenging task that enables the evaluation of agentic SE capabilities from a unique perspective.

5.2 RQ2: Performance on Doc2Feat-Bench Full

Considering the better average performance of Agentless and the high cost of OpenHands, we use Agentless to evaluate the models on Doc2Feat-Bench Full, and report the results in Table 5. We find that all four models show decreased performance on Doc2Feat-Bench Full compared to Doc2Feat-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 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 documentation changes or overly specific tests) in Doc2Feat-Bench Full. Since Doc2Feat-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, Doc2Feat-Bench Full may contain some noise, including unclear documentation changes or low-coverage regression tests, which make it more representative of real-world documentation-driven feature addition tasks, but harder for agents to solve. These factors make it more challenging.

It is worth noting that the ranking and relative performance of the models on Doc2Feat-Bench Verified are preserved on Doc2Feat-Bench Full except for Qwen3-235B and GPT-4o. 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 Doc2Feat-Bench Verified, while GPT-4o outperforms Qwen3-235B on Doc2Feat-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. 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.

RQ2 Summary: On Doc2Feat-Bench Full, DeepSeek-R1 achieves the best success rate of 14.83%. The average success rate drops from 18.20% on Doc2Feat-Bench Verified to 10.65% on Doc2Feat-Bench Full, showing that Doc2Feat-Bench Full is substantially more challenging. Most models preserve similar relative rankings across the two datasets, indicating broadly consistent capabilities across cleaner and more realistic settings.

Table 6: Performance across different software engineering agents on Doc2Feat-Bench for *single-file* and *multi-file* tasks.

Model	Scaffold	Single-File Modification		Multi-File Modification	
		Success%	Applied%	Success%	Applied%
Doc2Feat-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)
Doc2Feat-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)

GPT-4o is a notable exception, showing stronger robustness by outperforming Qwen3-235B on Doc2Feat-Bench Full despite similar performance on Doc2Feat-Bench Verified.

5.3 RQ3: Failure Analysis

To investigate why agents fail to solve tasks on Doc2Feat-Bench, we manually examine 160 failed instances (randomly sampling 10 instances per model per scaffold, i.e., 10 × 2 × 8 = 160) and identify the following primary reasons for failures on Doc2Feat-Bench through a qualitative analysis of execution traces.

Lack of Cross-File Editing Capability. In Doc2Feat-Bench Verified and Doc2Feat-Bench Full, only 53.5% and 45.7% of the instances can be solved by editing a single file, respectively, indicating that Doc2Feat-Bench requires strong cross-file editing capability. 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 software engineering agents on *single-file* and *multi-file* tasks, and present the results in Table 6. On Doc2Feat-Bench Verified and Doc2Feat-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 failures on Doc2Feat-Bench.

Lack of Comprehensive Understanding of Existing Code Modules. Through manual inspection, we find that agents often attempt to implement new features by directly modifying the code of existing features, and 39.38% of the examined instances exhibit

```
"""DeepSeek-v3 Patch"""
# /pylint/reporters/text.py
class ColorizedTextReporter(TextReporter):
-    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"""
# /django/forms/models.py
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)
```

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

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*". 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 F2P tests, it results in numerous regression test failures. This example demonstrates how agents may lack understanding of existing code modules needed to extend functionality without breaking existing features. Users may alleviate such phenomena by providing the agent with information about the differences between existing modules or by guiding the agent to analyze existing modules through a predefined plan.

Documentation Semantic Misinterpretation. To understand how models misinterpret documentation semantics, we identify three of the most frequent misinterpretation patterns during manual inspection, and illustrate each of them with a representative case in Figure 6. ❶ The model misunderstands when/how exceptions should be raised or suppressed (33.13%). For example, in *astropy-11691*, the documentation requires invalid attribute-style access to follow Python's module-level `__getattr__` behavior [34] and raise `AttributeError`, while the model only extends name-based lookup logic and fails to fully preserve the intended exception semantics. ❷ The model misinterprets edge conditions, thresholds, or conditional clauses (20.00%). For example, in *astropy-11954*, the documentation specifies strict constraints on `keys_left`/`keys_right`, such as mutual exclusivity with `keys` and consistency requirements, but the

Golden Patch	Model Patch

`+ def __getattr__(name):`
`+  if name in _attrname_physical_mapping:`
`+  return _attrname_physical_mapping[name]`
`+  raise AttributeError(...)`
	`+ if name in _name_physical_mapping:`
	`+  return _name_physical_mapping[name]`
	`+ elif name in _attrname_physical_mapping:`
	`+  return _attrname_physical_mapping[name]`
	astropy-11691

`+ if keys_left is None or keys_right is None:`
`+  raise ValueError('keys_left and keys_right must both be provided')`
`+`
`+ if keys is not None:`
`+  raise ValueError('keys arg must be None if keys_left and keys_right are supplied')`
	`+ left_copy = left.copy(copy_data=False)`
	`+ right_copy = right.copy(copy_data=False)`
	`+`
	`+ left_copy[temp_key] = left_key_data`
	`+ right_copy[temp_key] = right_key_data`
	astropy-11954

`+ if isinstance(n, str) and n.lower() == 'auto':`
`+  return n`
`+ n = _validate_int_greaterequal0(n)  # >= 0`
	`+ val = validate_int(s)`
	`+ if val <= 0:`
	`+  raise ValueError(...) # requires > 0`
	matplotlib-18715

Figure 6: Representative cases of documentation semantic misinterpretation.

model mainly treats the feature as a temporary-key transformation and does not fully implement the documented parameter contract. ❸ The documentation does not specify how to handle invalid input, and the implementation generated by the model differs from that of the developers (7.88%). For example, in *matplotlib-18715*, although the documentation requires the new configuration option to preserve the original behavior when set to `auto`, the model changes the validation boundary and thus deviates from the developers' intended default behavior.

> **RQ3 Summary:** Failures on Doc2Feat-Bench mainly stem from three issues. First, agents struggle with cross-file editing, leading to very low success rates on multi-file tasks. Second, they often lack a clear understanding of existing code modules and may break prior functionality when adding features. Third, they frequently misinterpret documentation semantics, such as exception handling and edge conditions.

6 Discussion

6.1 Doc2Feat-Bench Verified vs. Full

The verification process in Section 2.2.3 shows that 124 of the 238 sampled candidates (52.1%) were filtered out, suggesting that about half of the instances in Doc2Feat-Bench Full may contain unclear task descriptions or inaccurate evaluations. These quality issues reflect the realistic and naturalistic nature of tasks mined from real repositories, as PRs are created for human collaboration and are often iteratively clarified through maintainer-contributor discussions. Retaining these instances in Doc2Feat-Bench Full allows it to preserve this naturalistic distribution. Specifically, Doc2Feat-Bench Full enables evaluating agents' or models' robustness to the noise and data-quality variation of real development data, and avoids the selection bias toward simpler tasks introduced by strict filtering. In addition, Doc2Feat-Bench Full maintains the validity of the evaluation, as model rankings remain largely consistent between the two sets (RQ2). In summary, the two sets provide different evaluation perspectives. Doc2Feat-Bench Verified is suited to controlled experiments, fair comparison across models, and evaluations that require high-quality annotations. In contrast, Doc2Feat-Bench Full is suited to studying the realistic distribution of development data and robustness to noise, and thus better reflects the effectiveness of agents in real-world development.

6.2 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, Doc2Feat-Bench Verified, to increase the reliability of the evaluation. *Data Leakage.* The instances in Doc2Feat-Bench are collected from historical commits in specific GitHub repositories. As GitHub data is widely used to train state-of-the-art models, 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 systems suggest the impact of data leakage is limited. In addition, as a sanity check, we follow the methodology in [4] and compute Levenshtein similarity between generated patches and ground-truth patches for all successful instances of the best-performing model (DeepSeek-R1) on Doc2Feat-Bench Full. The average similarity is only 0.48, suggesting systems are not simply copying memorized patches.

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 Doc2Feat-Bench is language-agnostic and can be adapted to other ecosystems in future work. *Scaffold Selection.* We evaluate the performance of the selected systems 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 models indicates that Doc2Feat-Bench poses new challenges for agentic software engineering methods.

7 Related Work

7.1 Benchmarks for Agentic SE Systems

Evaluating the capabilities of agentic systems in software engineering (SE) has attracted increasing attention, leading to the development of several benchmarks. Among these, SWE-bench [23] has emerged as one of the most influential benchmarks for assessing systems on issue resolution tasks. A line of follow-up work has extended SWE-bench to multilingual [50, 51], multimodal [19, 46], and industrial [37, 48] scenarios, promoting broader language and scenario coverage in issue-solving tasks. FEA-Bench [25] aims to evaluate systems on feature addition tasks, with detailed code-level specifications, docstrings, and pull request descriptions as task inputs. However, these benchmarks rely on maintainer-facing inputs, such as issue reports, and assume familiarity with the codebase and implementation-level terminology [13, 15]. As a result, they provide limited insight into capabilities under public-facing specification inputs. In addition, most of them focus on bug fixing or issue solving, while largely overlooking feature addition, which accounts for about 60% of software maintenance activities [35].

Table 7 provides a detailed comparison between Doc2Feat-Bench and related benchmarks, highlighting several distinctive features of Doc2Feat-Bench: (1) Doc2Feat-Bench adopts a documentation-driven setting where public-facing documentation changes are provided as inputs; (2) Doc2Feat-Bench does not constrain instances to those introducing new components, recognizing that many feature additions extend existing code; (3) Doc2Feat-Bench identifies feature-adding PRs based on developer-maintained release notes to reduce noise [44, 52]. In addition, we manually construct a verified subset to enable lightweight yet reliable evaluation.

Table 7: Comparison with existing benchmarks.

Aspect	SWE-bench	GITS-Eval	FEA-Bench	Doc2Feat-Bench
Development scenario	Maintainer-facing	Maintainer-facing	Maintainer-facing	Public-facing
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 models	Mining release notes
Verified subset	Yes	Yes	No	Yes
Open-source	Yes	No	Yes	Yes

7.2 Automating SE Tasks with Agentic Methods

Recently, agentic methods [21, 22, 27, 41] 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 an underlying model to follow a fixed workflow. Examples include Agentless [43], and PatchPilot [24]. Agent-based methods equip agents with various tools (e.g., bash and language servers) to access and modify the codebase, and rely on model-driven decision-making and tool invocation. Examples include OpenHands [42], RepairAgent [11], and ExecutionAgent [12]. While these methods have achieved significant advances, they are typically evaluated in a maintainer-facing way, where the inputs include implementation-level details and assume programming expertise. In contrast, feature addition tasks in Doc2Feat-Bench are specified with public-facing documentation changes. This mismatch introduces new and unique challenges for applying prior methods to Doc2Feat-Bench.

8 Conclusion

We introduce Doc2Feat-Bench, a challenging benchmark for evaluating software engineering capabilities in documentation-driven feature addition. Doc2Feat-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, Doc2Feat-Bench requires systems to generate a patch that adds a new feature. We evaluate 8 advanced models using two representative scaffolds, Agentless and OpenHands. Our results show that documentation-driven feature addition is a uniquely difficult and unsolved problem. Moreover, we analyze failure cases on Doc2Feat-Bench to guide future improvements in cross-file editing, code module understanding, and documentation semantic interpretation.

Data Availability

The replication package and online Appendix are available at https://doi.org/10.5281/zenodo.21762257. Our official repository is available at https://github.com/Doc2Feat-bench/Doc2Feat-bench. The leaderboard of Doc2Feat-Bench is available at http://doc2featbench.org. An earlier version of Doc2Feat-Bench was referred to as NoCode-bench.

Acknowledgments

This research is supported by the National Natural Science Foundation of China (No.92582107), Zhejiang Provincial Natural Science Foundation of China (No.LZ25F020003), and by the German Research Foundation (DFG; projects 492507603, 516334526, and 526259073).

References

[1] 2001. IEEE Standard for Software User Documentation. *IEEE Std 1063-2001* (2001), 1–24.
[2] 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*. 1199–1210.
[3] Halim Ahmad and Hasnita Halim. 2017. Determining sample size for research activities: the case of organizational research. *Selangor Business Review* (2017), 20–34.
[4] 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*.
[5] Bartosz Alchimowicz and Jerzy R Nawrocki. 2016. The COCA quality model for user documentation. *Software quality journal* 24, 2 (2016), 205–230.
[6] Anthropic. 2024. Claude 4. https://www.anthropic.com/news/claude-4. Accessed: 2025-01.
[7] Keith H Bennett and Václav T Rajlich. 2000. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. 73–87.
[8] Aaditya Bhatia, Shaowei Wang, Muhammad Asaduzzaman, and Ahmed E Hassan. 2020. A study of bug management using the Stack Exchange question and answering platform. *IEEE Transactions on Software Engineering* 48, 2 (2020), 502–518.
[9] 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.
[10] 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*. 188–197.
[11] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *International Conference on Software Engineering*.
[12] 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*.
[13] 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*. 86–96.
[14] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261* (2025).
[15] GitHub, Inc. 2025. GitHub Issues · Project planning for developers. https://github.com/features/issues.
[16] GitHub, Inc. 2025. GitHub Spec Kit: Toolkit to help you get started with Spec-Driven Development. https://github.com/github/spec-kit.
[17] R.L. Glass. 2001. Frequently forgotten fundamental facts about software engineering. *IEEE Software* 18, 3 (2001), 112–111.
[18] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).
[19] 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. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 24–46.
[20] Abram Hindle, Daniel M German, and Ric Holt. 2008. What do large commits tell us? a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories*. 99–108.
[21] Zhonghao Jiang, David Lo, and Zhongxin Liu. 2025. Agentic Software Issue Resolution with Large Language Models: A Survey. *arXiv preprint arXiv:2512.22256* (2025).
[22] Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. 2025. Issue Localization via LLM-Driven Iterative Code Graph Searching. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering*. 3034–3045.
[23] 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*.
[24] Hongwei Li, Yuheng Tang, Shiqi Wang, and Wenbo Guo. 2025. PatchPilot: a cost-efficient software engineering agent with early attempts on formal verification. In *Proceedings of the 42nd International Conference on Machine Learning*.
[25] 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. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 17160–17176.
[26] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).
[27] Noor Nashid, Islem Bouzenia, Michael Pradel, and Ali Mesbah. 2025. Issue2Test: Generating Reproducing Test Cases from Issue Reports. *arXiv preprint arXiv:2503.16320* (2025).
[28] OpenAI. 2024. GPT-4o System Card. *arXiv preprint arXiv:2410.21276* (2024).
[29] OpenAI. 2024. SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/.
[30] OpenAI. 2025. Introducing GPT-5. https://openai.com/index/introducing-gpt-5/. Accessed: 2025-10-17.
[31] OpenAPI. 2025. Usage | OpenAPI Generator. https://openapi-generator.tech/docs/usage.
[32] Pallets. 2025. Flask: Micro Framework for Building Web Applications. https://flask.palletsprojects.com/en/stable/.
[33] Tom Preston-Werner. 2010. Readme Driven Development. https://tom.preston-werner.com/2010/08/23/readme-driven-development.
[34] Python Software Foundation. 2026. The Python Language Reference: Data Model. https://docs.python.org/3/reference/datamodel.html.
[35] 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*. 226–229.
[36] 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 (2016), S3–S11.
[37] 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*.
[38] Mozhan Soltani, Felienne Hermans, and Thomas Bäck. 2020. The significance of bug report elements. *Empirical Software Engineering* 25, 6 (2020), 5255–5294.
[39] Ian Sommerville. 2015. *Software Engineering*.
[40] SymPy Development Team. 2025. SymPy: Python Library for Symbolic Mathematics. https://docs.sympy.org/latest/index.html.
[41] Junyi Wang, Jialun Cao, and Zhongxin Liu. 2026. iCoRe: An Iterative Correlation-Aware Retriever for Bug Reproduction Test Generation. *Proceedings of the ACM on Software Engineering* (2026), 4231–4252.
[42] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2025. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*. 65882–65919.
[43] 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.
[44] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. 2024. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817* (2024).
[45] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).

[46] 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*.

[47] 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. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

[48] Zhifan Ye, Jiachi Chen, Zhenzhe Shao, Lingfeng Bao, Xiaohu Yang, and Zhongxin Liu. 2025. SolContractEval: A Benchmark for Evaluating Contract-Level Solidity Code Generation. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering*. 2858–2869.

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

[50] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, et al. 2026. Multi-swe-bench: A multilingual benchmark for issue resolving. *Advances in Neural Information Processing Systems* 38 (2026).

[51] 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).

[52] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2025. Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models. *Computational Linguistics* (2025).

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