SmellBench: Towards Fine-Grained Evaluation of Code Agents on Refactoring Tasks

Anonymous ACL submission

Abstract

The growing capability of code agents has motivated a strong demand for comprehensive benchmarks to evaluate their software engineering abilities. However, current benchmarks mainly emphasize functional correctness, while overlooking the code maintainability, which leads to severe code bloat and structural degradation (i.e., code smells). Consequently, there is an urgent need for a benchmark dedicated to comprehensively assessing the refactoring capabilities of code agents to ensure code quality and maintainability. In this paper, we propose SmellBench, the first extensible code refactoring benchmark that proactively injects code smells into clean code snippets from real-world repositories. This design enables the generation of controlled, high-quality, and diverse refactoring cases with human-written ground truth. Specifically, it contains 294 cases spanning 7 common smell types, 3 difficulty levels, 2 instruction settings across 7 real-world repositories. We further design 3 evaluation aspects covering functional correctness, localization ability, and refactoring quality assessment. Experiments with 2 popular agents and 6 large langauge models (LLMs) show that the best combination—Qwen Code + Claude Sonnet 4.5—achieved only a 50.34 score of refactoring quality. Analysis reveals that this gap arises from a focus on local code smells and a lack of cross-file understanding, which hinders comprehensive smell elimination. Our code and benchmark are released at https://anonymous.4open.science/r/SmellBench-5602/

1 Introduction

Recent advances in large language models (LLMs) have sparked the development of powerful code agents capable of solving complex software engineering tasks, including code generation, bug fixing, and repository-level reasoning. (Wang et al., 2024; Yang et al., 2025, 2024; Aider, 2023). To systematically evaluate these capabilities, a growing number of benchmarks have been proposed for software engineering agents (Jimenez et al., 2023; Merrill et al., 2026; Jain et al., 2024; Zan et al., 2025). However, existing benchmarks mainly focus on functional correctness-oriented tasks such as code generation and bug fixing, while paying limited attention to software maintainability and structural quality, which may result in code bloat and structural degradation. Recently, code refactoring has garnered much attention as a critical capability, where the task aims at improving existing code quality, enhancing maintainability, and supporting the long-term evolution of software systems (Liu et al., 2025b; Gautam et al., 2025; Garg et al., 2025).

Existing refactoring benchmarks can be broadly divided into two lines. One line of research focuses on curating refactoring-oriented benchmarks (Chen, 2021; Gautam et al., 2025; Jain et al., 2024; Tapader et al., 2025), which offer high-quality refactoring cases with clear instructions. However, the substantial human effort leads to unaffordable cost and limited scalability, which in turn restricts the coverage of diverse refactoring scenarios in real-world codebases. Another line of benchmark automatically mines refactoring cases from the commit histories of GitHub repositories(Xu et al., 2026; Garg et al., 2025; Pomian et al., 2024; Kovacic et al., 2025). Although this approach offers better scalability, its reliance on mining strategies leads to an emphasis on relatively simple and surface-level refactoring types, such as code movement and local modifications, while more complex refactoring behaviors are largely underexplored. A key reason for this gap lies in the nature of real refactoring behavior. Refactoring rarely occurs as a standalone activity. Instead, it is typically distributed across multiple commits and tightly intertwined with feature implementation and bug fixing. As a result, refactoring signals in real repositories are sparse,

fragmented, and unevenly distributed, making it difficult for commit-history–based benchmarks to systematically capture diverse and realistic refactoring scenarios.

To address this limitation, we propose **SmellBench**, the first extensible code refactoring benchmark that *proactively injects code smells into clean code snippets from real-world repositories*. This design supports the generation of controlled, high-quality, and diverse refactoring cases with human-written ground truth. SmellBench contains 294 cases spanning 7 smell types, 3 difficulty levels and two instruction settings across 7 real-world repositories to ensure comprehensive evaluation of refactor ability of current code agents. SmellBench contains 294 cases spanning 7 smell types, 3 difficulty levels and two instruction settings across 7 real-world repositories to ensure comprehensive evaluation of refactor ability of current code agents. Overall, our SmellBench offers the following advantages: (1) **Purity**. It isolates refactoring from other code changes. (2) **Diversity**. It covers 7 complex smell types. (3) **Authenticity**. It provides human-written ground truth. (4) **Extensibility**. It can be easily extended to other smell types or even task types by introducing new specifications. (5) **Decontamination**. It avoids data leakage with newly generated cases.

We further design 3 evaluation aspects specifically for refactoring tasks, including test passing rate to ensure functional correctness, localization accuracy to assess the identification of refactoring targets, and LLM-based judging score to evaluate the refactoring quality. Moreover, we conduct experiments on 2 open-source code agents and 6 representative LLMs. The results show that the best-performing configuration, Qwen Code with Claude Sonnet 4.5, achieves a refactoring quality score of only 50.34, indicating that existing approaches mainly focus on local code smells and struggle to identify and address smells that propagate across multiple files. Meanwhile, most models pass over 80% test cases, suggesting that relying solely on functional correctness is insufficient to effectively distinguish refactoring capabilities. This further highlights the necessity of introducing LLM-as-Judge for refactoring quality assessment.

In summary, we highlight the main strengths of this work as follows:

- We propose **SmellBench**, the first extensible code refactoring benchmark that proactively injects code smells into clean code snippets, bridging the trade-off between data scalability and quality in existing benchmarks.

- SmellBench comprises 294 high-quality refactoring cases spanning 7 complex smell types, 3 difficulty levels, and 2 instruction settings across 7 real-world repositories, with human-written ground truth for evaluation.

- We design a 3-dimensional evaluation framework tailored for refactoring tasks, including test passing rate, localization accuracy, and LLM-based quality assessment, providing more nuanced insights beyond functional correctness.

- Extensive experiments with 2 open-source agents and 6 representative LLMs reveal that current approaches achieve only 55.86 in refactoring quality, struggling particularly with cross-file smell identification, which highlights significant room for improvement.

2 Related Work

2.1 Code Benchmark

Benchmarking the coding capabilities of large language models has evolved with the increasing complexity of code generation tasks(Chen, 2021; Austin et al., 2021; Jimenez et al., 2023; Jain et al., 2024; Ni et al., 2025; Zhuo et al., 2024). Early benchmarks such as HumanEval (Chen, 2021) and MBPP (Austin et al., 2021) focus on isolated programming problems and measure the functional correctness of generated code at the function or algorithm level. Beyond standalone functions, ClassEval (Du et al., 2023) targets class-level code generation, and repository-oriented benchmarks including RepoBench (Liu et al., 2023) and CrossCodeEval (Ding et al., 2023) extend the scope to cross-file and project-level code modeling, emphasizing contextual reasoning in realistic codebases. Subsequently, research attention shifted toward real-world software engineering problems. SWE-bench (Jimenez et al., 2023) centers on bug fixing derived from real GitHub issues. LiveCodeBench (Jain et al., 2024) draws tasks from contests across three competitive programming platforms, placing greater demands on existing large language models. More recently, task-oriented benchmarks have been proposed to assess code agents under realistic

user demands. GitTaskBench (Ni et al., 2025) anchors benchmarking in repository-based tasks, and CodeIF (Yan et al., 2025) characterizes instruction-following behavior across diverse code generation scenarios, shifting the paradigm toward user-driven, multi-step problem solving.

However, these benchmarks focus mainly on code generation and bug fixing, and pay limited attention to refactoring tasks that are fundamental and critical to software maintenance and long-term evolution.

2.2 Refactor Benchmark

Alongside the rapid progress of code generation benchmarks, research on evaluating code refactoring has gained increasing attention in recent years (Gautam et al., 2025; Xu et al., 2026; Liu et al., 2025b; Kuang Piao et al., 2025). Early work constructs refactoring datasets from a small number of projects, mostly in Java, and evaluates LLMs on identifying refactoring opportunities and applying corresponding code changes (Liu et al., 2025b; Tapader et al., 2025). These studies show that prompt design can significantly affect refactoring performance, but their primary goal is to analyze LLM behavior instead of modeling realistic refactoring workloads. Later efforts extend refactoring to more complex settings, such as stateful refactoring across multiple steps (Gautam et al., 2025), extracting shared logic into reusable libraries (Kovacic et al., 2025), or combining LLM suggestions with IDE or static analysis tools (Pomian et al., 2024). These directions emphasize broader code context and design-level reasoning. More recent benchmarks attempt to leverage real-world repositories by mining refactoring-related commits or covering a wider range of refactoring types (Kuang Piao et al., 2025; Xu et al., 2026). However, these approaches often rely heavily on external refactoring detection tools or curated examples, limiting scalability and language coverage. Related work also studies specialized refactoring scenarios (Garg et al., 2025), such as resolving performance-related code issues, which can be viewed as a task-specific form of refactoring.

Overall, existing refactoring benchmarks either rely on manually constructed datasets that are costly and limited in scale, or mine refactoring instances from repository commits that inevitably mix refactoring with other code changes, leading to limited coverage, scalability, or data purity.

3 SmellBench

This section is organized as follows: First, we provide a brief description of the 7 smell types. Next, we present the construction pipeline of the benchmark, which consists of four steps. Finally, we describe the evaluation process and the 3 metrics.

3.1 Smell Types

Code smells are a long-established abstraction in software engineering for characterizing structural design deficiencies that motivate refactoring. Importantly, they naturally encode refactoring objectives that go beyond local edits, often involving interactions across functions, classes, and files. By grounding refactoring tasks in a curated set of representative code smell types, we can construct refactoring scenarios that are both structurally meaningful and amenable to controlled evaluation.

Specifically, we instantiate refactoring tasks using 7 representative code smell types. Their formal definitions refer to Figure 12 and illustrative toy examples are provided in Figure 1.

3.2 Benchmark Construction

The construction of SmellBench follows four steps: collecting real-world software repositories, identifying suitable injection locations, introducing code smells into clean code, and validating functional correctness using test cases. The overall workflow is illustrated in Figure 3.

3.2.1 Repository Selection

We select 7 Python repositories from top-ranked PyPI projects as our codebases. Popular, large-scale repositories are preferred due to their mature contributor guidelines, more comprehensive test coverage, and complex code dependencies.

3.2.2 Candidate Discovery

Following repository collection, we identify appropriate locations for injecting specific code smells. This stage is designed to ensure both coverage and validity of injection locations. First, real-world software repositories are typically large and heterogeneous. Allowing a code agent to search for injection locations without constraints can lead to location bias, where the agent disproportionately selects candidates from early-accessed files while neglecting other suitable regions of the codebase. Second, code smells are not universally applicable. Each smell type imposes distinct structural or semantic requirements, meaning that only a subset

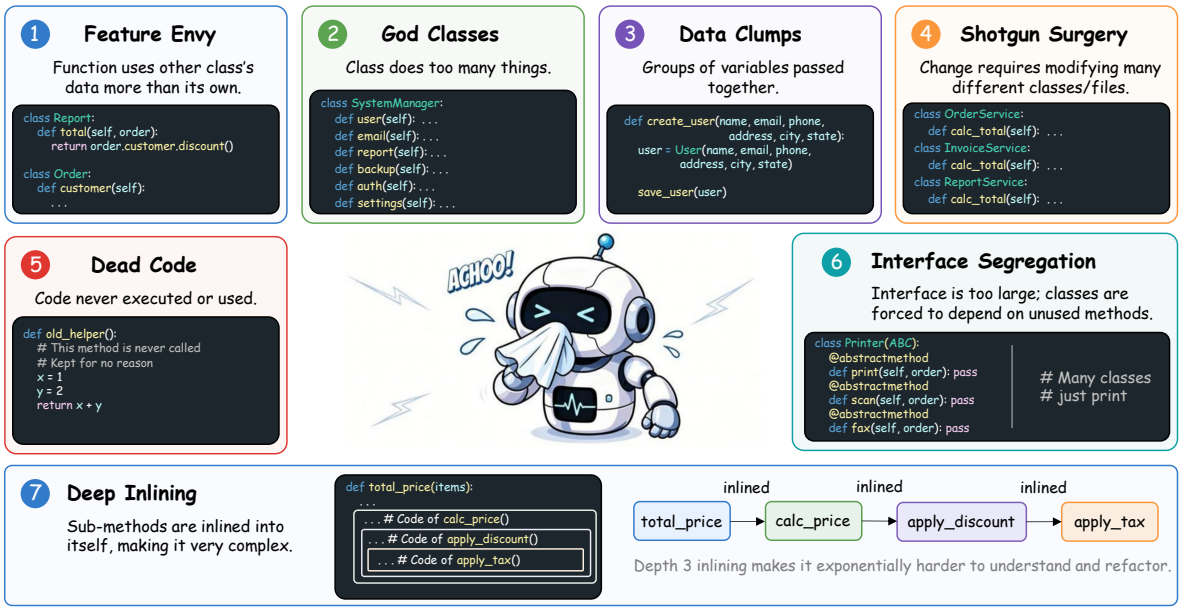

Figure 1: Definitions of the 7 code smell types with illustrative toy examples.

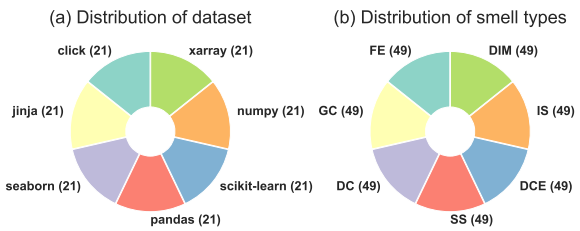

Figure 2: Distribution of SmellBench across 7 python repositories and 7 smell types.

of functions or classes constitute valid injection locations. Consequently, candidate selection must be guided by smell-specific constraints rather than unrestricted search. To address these challenges, Candidate Discovery is decomposed into two submodules:

Coarse Filtering. This module reduces the search space by pre-selecting files that are more likely to contain meaningful injection targets. In our current setup, we retain only source files exceeding 500 lines of code, which tend to exhibit higher structural complexity and are therefore more suitable for smell injection.

Agent Localization. Given the filtered files, a code agent is used to identify candidates that satisfy the requirements of the smell specification. This step focuses exclusively on locating and selecting candidates and does not perform any code modifications. The resulting candidate set is passed to the subsequent smell injection stage.

3.2.3 Smell Injection

After obtaining clean code candidates, this stage automatically constructs code samples containing target code smells. Specifically, the code agent is fed with candidates and the smell specification, which directly modifies the original code to introduce the target smell while preserving compilability.

In contrast to prior methods, this stage relies solely on a sufficiently powerful code agent (e.g., OpenHands + Claude Sonnet 4.5), and is therefore frees from programming language, external tools or any commit histories. As a result, SmellBench is theoretically capable of producing unlimited number of cases via injecting various smells across different candidates, which overcomes the challenges of scalability and diversity arising from commit-based approaches.

To facilitate a systematic analysis of agent behavior, we further introduce 3 levels of case difficulty and 2 types of instruction settings. These configurations allow us to examine how code agents respond to varying levels of task complexity and instructional constraints. Detailed definitions and examples of these settings are provided in Appendix D and E.

Finally, we retain both the original and modified versions of the code, forming paired clean–smelly samples that serve as the basis for subsequent quality verification and evaluation.

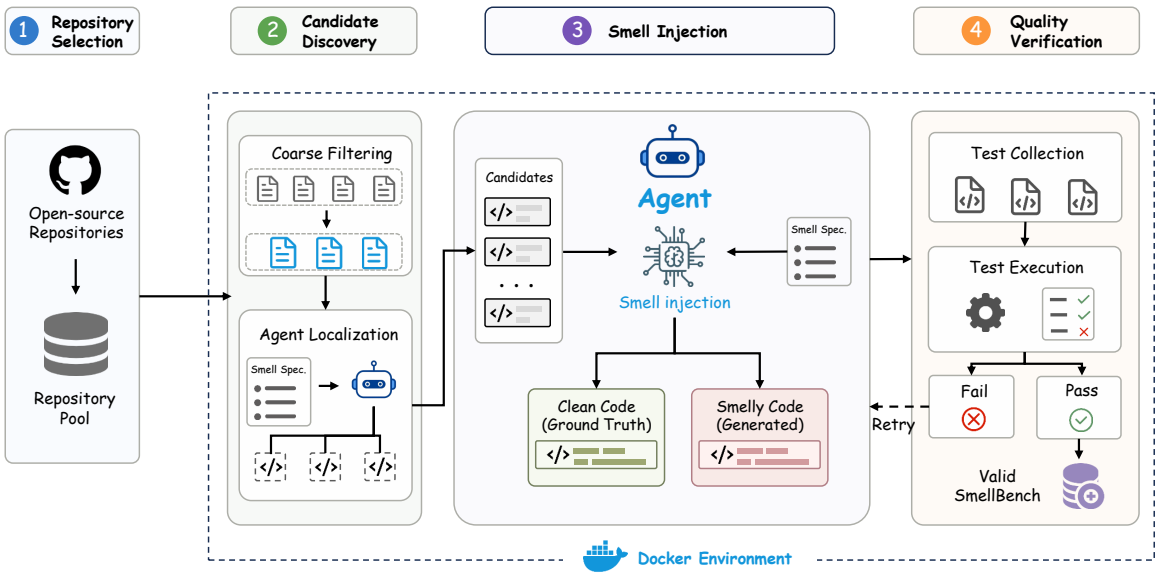

Figure 3: SmellBench construction pipeline: Repository Selection to collect repositories, Candidate Discovery to identify suitable injection locations, Smell Injection to introduce controlled code smells, and Quality Verification to validate correctness and benchmark reliability. The latter three stages are executed in a Dockerized environment.

3.2.4 Quality Validation

Recall that refactoring tasks aim to improve code structure while preserving program semantics and utility. Therefore, we are supposed to ensure functional correctness of smelly code, i.e., passing the unit tests.

To this end, we collect test suites related to the smelly code and execute the associated tests. Specifically, coverage analysis is employed to determine which test cases invoke the methods contained in the smelly code, with each method being associated with at most 10 relevant test cases. Test-failing cases, along with their error messages, are fed back to the code agent for rectification, thereby enhancing both pass rates and generation efficiency. Finally, cases are filtered out if no relevant tests are available or if the tests fail again.

This test-based validation process allows SmellBench to retain high-quality and non-trivial refactoring cases, supporting a comprehensive assessment of code agent refactoring capabilities.

After these four steps, we finally generate 294 valid samples with 7 smell types, 3 difficulty levels and 2 instruction settings across 7 repositories. The statistics of the SmellBench are illustrated in Figure 2, where each category is evenly distributed, demonstrating the controllability of the construction pipeline.

3.3 Evaluation Framework

To comprehensively assess code agents in refactoring tasks, we examine not only whether their changes pass existing tests but also the overall quality of the refactoring in terms of structural soundness and smell elimination. Specifically, our evaluation consists of three aspects, including test passing rate to ensure functional correctness, localization accuracy to assess the identification of refactoring targets, and LLM-based judging score to evaluate the refactoring quality.

Test Pass Rate. Test pass rate (TPR) measures whether the code refactors preserve the original program behavior.

Localization Accuracy. Localization accuracy (LA) evaluates whether a code agent correctly identifies the code locations that require refactoring.

LLM-as-Judge. LLM-as-Judge employs LLMs to assess refactoring quality from four complementary aspects: *code quality* (CQ), which measures readability and maintainability improvements; *structural soundness* (SS), which evaluates the correctness and rationality of the refactored structure; *cross-file coordination* (CFC), which assesses consistency and dependency handling across multiple files; and *smell elimination* (SE), which measures the effectiveness of removing the targeted code smells. The detailed rubric information can

Table 1: Performance comparison of different agents and LLMs on SmellBench. TPR represents the Test Pass Rate, LA denotes the Localization Accuracy, CQ is code quality, SS stands for structural soundness, CC refers to cross-file coordination, SE indicates smell elimination. The bold numbers indicate the best results.

Agent	LLM	TPR	LA	LLM-as-Judge			
				CQ	SS	CC	SE
Oracle	-	1.0	1.0	0.9592	0.9626	0.9884	0.9660
OpenHands	Qwen-Coder-30B-A3B	0.8021	0.7067	0.4649	0.4110	0.3014	0.3004
	Qwen-Coder-480B-A35B	0.8258	0.7456	0.5647	0.5084	0.3962	0.4080
	DeepSeek-V3.2	0.8790	0.7224	0.5801	0.5203	0.3804	0.3733
	gemini-2.5-flash	0.5272	0.6667	0.4041	0.3592	0.2323	0.2327
	gpt-5-mini	0.9286	0.7721	0.6146	0.5418	0.4316	0.4235
	claude-sonnet-4.5	0.9116	0.8401	0.6779	0.6139	0.4881	0.4793
Qwen Code	Qwen-Coder-30B-A3B	0.7789	0.6701	0.4203	0.3574	0.2344	0.2251
	Qwen-Coder-480B-A35B	0.8197	0.6531	0.4573	0.3990	0.2812	0.2867
	DeepSeek-V3.2	0.8571	0.4966	0.3677	0.3248	0.2405	0.2503
	gemini-2.5-flash	0.6633	0.6395	0.4163	0.3735	0.2231	0.2279
	gpt-5-mini	0.9218	0.7483	0.5626	0.4871	0.3238	0.3255
	claude-sonnet-4.5	**0.9388**	**0.8878**	**0.6929**	**0.6405**	**0.5143**	**0.5034**

be found in Appendix F.

In LLM-as-Judge part, the LLM conducts a comparative analysis between the code smells and the refactoring results from agent, and derives an evaluation judgment. However, this process demands strong contextual understanding and reasoning capabilities from the LLM to disentangle the structural issues and logical relationships implicitly embedded in complex code smells, which can easily lead to inconsistent evaluation outcomes across different LLMs. To enable stable and consistent conclusions among different models, we introduce a preliminary step, Smell Analysis.

Smell Analysis. Smell analysis explicitly analyzes and summarizes the code smells to clarify the corresponding structural issues and refactoring objectives, and provides clear and unified contextual grounding for subsequent LLM-based evaluation. Please refer to Figure 12 for detailed information.

Notably, SmellBench has been integrated with the Harbor framework (Harbor Framework Team, 2026), which performs evaluation within Docker environments and supports a wide range of popular code agents.

4 Experiments

4.1 Experimental Settings

To examine whether existing code agents can effectively address our benchmark, we select some of the most popular and powerful agents together with representative LLMs. Specifically, we choose two widely used open-source agents, OpenHands (Wang et al., 2024) and Qwen Code (Yang et al., 2025), paired with six large language models (LLMs), including three cutting-edge open-source models — Qwen3-Coder-30B-A3B-Instruct (Yang et al., 2025), Qwen3-Coder-480B-A35B-Instruct (Yang et al., 2025), and DeepSeek-V3.2 (Liu et al., 2025a) — and three closed-source models — GPT5-mini (Achiam et al., 2023), Gemini2.5-flash (Comanici et al., 2025), and Claude Sonnet-4.5 (Anthropic, 2025). The maximum evaluation time is 20 minutes.

4.2 Overall Performance

We report the overall performance in Table 1, which presents the code refactoring capabilities evaluated across two open-source agents and six large language models (LLMs). From the results, we obtain the following key observations. (1) **Refactoring task is still challenging for SOTA baselines**. Even the best-performing baseline, Qwen-Code + Claude Sonnet 4.5, only achieved a Smell Elimination (SE) score of 0.5034, highlighting the significant gap between current code generation capabilities and the deeper structural reasoning required for high-quality refactoring. (2) **Test Passing Rate is not suitable for refactoring task.** Except for Gemini 2.5 Flash, all other

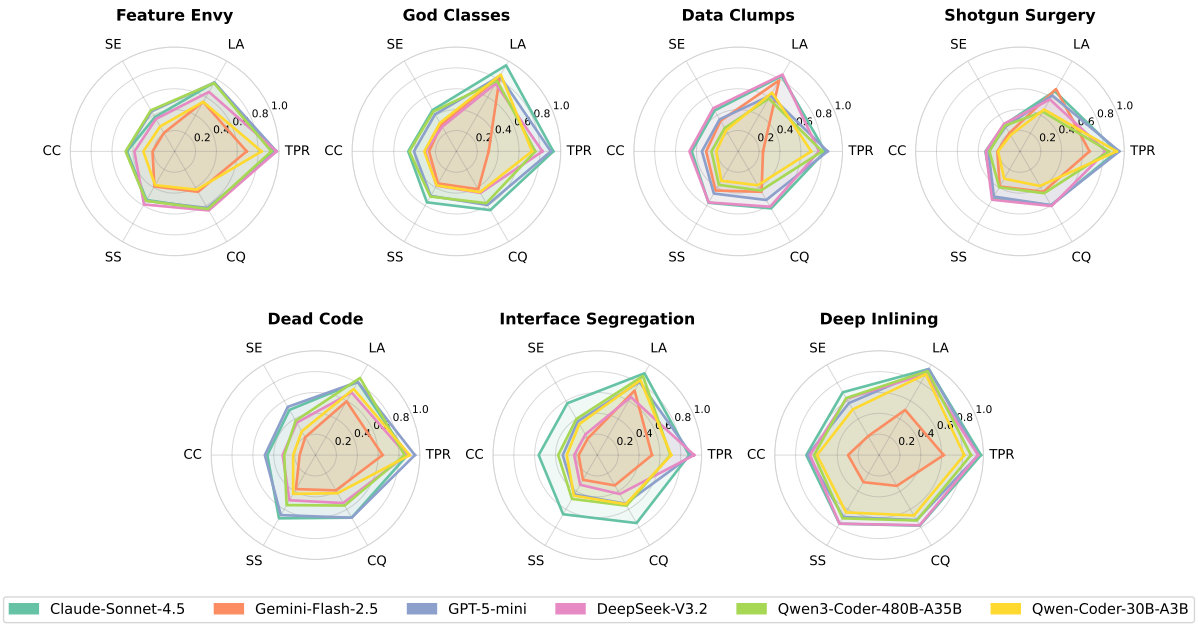

Figure 4: Performance comparison of 7 smell types under different LLMs with the OpenHands Agent. 6 metrics are adopted for comprehensive evaluation.

models achieve TPRs above 80%, indicating that current models place strong emphasis on preserving functional correctness. In addition, the scores for Code Quality and Structural Soundness (SS) are also relatively high, further reinforcing this observation. (3) **Open-source models also exhibit strong competitiveness in code refactoring tasks.** DeepSeek-V3.2 achieves remarkable scores of 37.37 and 37.12. This demonstrates the rapid advancement of open-source LLMs and their potential to rival closed-source models in complex code understanding and refactoring tasks. (4) **OpenHands demonstrates better performance than Qwen Code when coupled with weaker LLMs**. However, as the LLM capacity increases, the performance gap between the two agents narrows. This suggests that OpenHands possesses stronger exploration capabilities, enabling more effective utilization of limited model capacity. When the LLM becomes more powerful, the upper bound of performance is constrained by the model's inherent ability, leading both agents to converge toward similar results.

4.3 Analysis of Smell Types

In this subsection, we further analyze the performance of the seven code smells under different model configurations. The experiments employ OpenHands as the agent framework and select six representative large language models as backends. We evaluate the results using six metrics to reveal the differences in how various LLMs handle structural code issues. Based on the results shown in Figure 4, we make the following observations: (1) **Different code smells exhibit significant performance variations across different LLMs.** It can be observed that model performance varies considerably depending on the type of code smell. For example, DeepSeek-V3.2 performs particularly well on Data Clumps, Shotgun Surgery, and Deep Inlining, while its performance on Interface Segregation is relatively weak. These results demonstrate that our benchmark is capable of effectively evaluating the comprehensive abilities of different models. (2)

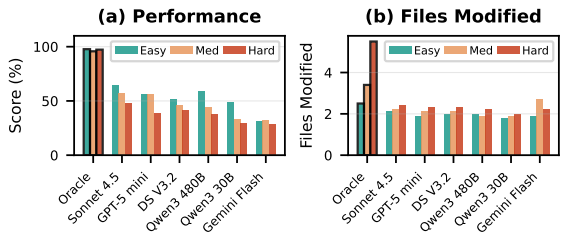

Figure 5: Performance comparison across difficulty levels. (a) scores (average of CQ, SS, CF, and SE) for Oracle and six models on Easy, Medium, and Hard tasks. (b) Number of files modified by each model.

All models perform poorly on Shotgun Surgery. Fixing Shotgun Surgery requires coordinated modifications across multiple files. Statistics show that

this type of smell involves an average of 5.7 files, which is significantly higher than the overall average of 2.5 files. However, Claude Sonnet 4.5 modifies only 2.7 files on average when addressing this smell. This suggests that current approaches still have substantial limitations in handling multi-file coordinated modifications. (3) **All models achieve strong performance on Deep Inlining.** Repairing this type of smell mainly relies on the model's ability to decompose code and perform deep reasoning. The experimental results indicate that all models perform well in these aspects, enabling them to effectively address this code smell.

4.4 Analysis of Difficulty Levels

This section mainly analyzes the impact of different difficulty levels on the number of task instances and model performance. By comparing the three difficulty levels—Easy, Medium, and Hard—we investigate the relationship between task difficulty and model capability. All experiments are conducted using OpenHands as the agent framework.

Based on the results shown in Figure 5, we draw the following conclusions: (1) As shown in Figure 5(a), the performance of all models gradually declines as the difficulty level increases, demonstrating the rationality of the benchmark's difficulty design. (2) Open-source models perform well on easy tasks but show relatively poor performance on difficult tasks, indicating that their deep reasoning capabilities still lag behind strong models such as Claude Sonnet 4.5. (3) Figure 5(b) shows that the number of involved files increases significantly as task difficulty rises. However, the number of modified files by all models does not increase correspondingly. This suggests that existing models, including Claude Sonnet 4.5, still exhibit substantial limitations in multi-file coordination.

4.5 Error Analysis

This section presents an analysis of model failure cases, which serve as a critical window for understanding model boundaries, facilitating targeted model selection while providing a more comprehensive perspective for the SmellBench evaluation framework. Specifically, we employed OpenHands as the agent, leveraging DeepSeek V3.2 as a LLM-based classification tool to perform fine-grained categorization of failure cases. This yielded six distinct error categories plus an "Other" category covering sparse error types. According to Figure 6, we have the following key findings: (1) AgentTimeout is a prominent issue. Severe timeout phenomena were observed across all models, reflecting the inherent complexity of the tasks—models require extended reasoning and multi-round revisions. Notably, DeepSeek exhibited a timeout rate as high as 77%. Further analysis revealed that the model engaged in excessively lengthy reasoning actions, rendering it unable to complete tasks within the limited time. (2) Bottleneck in localization capabilities. Location Failure accounts for a substantial proportion of errors across all models. This indicates that current models still face challenges in precisely localizing the positions of code smells. (3) Insufficient repair and coordination capabilities. Wrong Fix and Incomplete errors both accounted for notable proportions. This suggests that while models have acquired method-level localization capabilities, there still remain substantial room for improvement in formulating concrete repair strategies that how to fix refactor cases and performing cross-file coordinated editing.

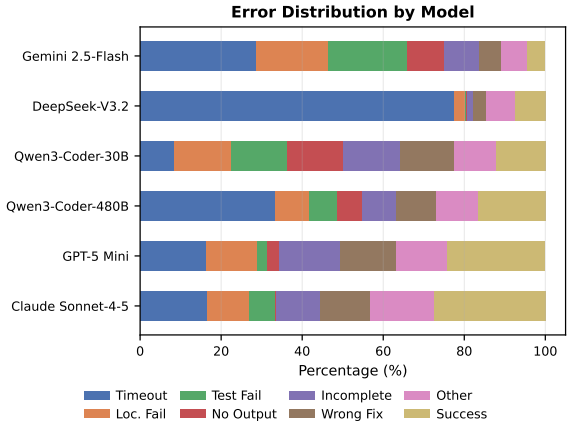

Figure 6: Error distribution of six LLM models on the SmellBench.

5 Conclusion

In this paper, we present **SmellBench**, the first extensible benchmark for repository-level code refactoring through proactive code smell injection. SmellBench contains 294 refactoring cases across 7 smell types and introduces a multi-dimensional evaluation framework beyond functional correctness. Experiments on 2 code agents and 6 representative LLMs show that current models still struggle with complex cross-file refactoring tasks, highlighting substantial gaps in repository-level reasoning and software maintainability capabilities.

Limitations

We acknowledge several limitations of SmellBench. (1) Since developers may not strictly comply with coding best practices, some code smells, such as God Classes, may naturally exist in the original repositories, which could introduce noise and reduce the reliability of the evaluation results. (2) The current smell injection process exhibits certain similarities across cases, it may encourage models to adopt similar solving patterns, which in turn reduces the diversity of SmellBench. (3) The use of LLM-as-Judge for evaluation introduces potential bias, as the assessment results may be influenced by the capability and inherent preferences of the judging model. This could affect the objectivity and stability of the evaluation outcomes.

Ethical Impact

In this paper, we introduce SmellBench, a benchmark designed to evaluate the code refactoring and maintainability capabilities of Code Agents. We recognize and explicitly address the following ethical implications associated with our work:

Data Privacy, Copyright, and Licensing. All source code and software projects included in SmellBench are constructed from [publicly available open-source repositories / synthetically generated pipelines / curated public datasets]. We have strictly adhered to the original licensing agreements (e.g., MIT, Apache 2.0) of all foundational codebeds. We conducted thorough data filtering to ensure that no Personally Identifiable Information (PII), proprietary vendor code, or sensitive data is included in our benchmark.

Code Safety and Dual-Use Risks. While evaluating and enhancing code refactoring capabilities promotes software maintainability and mitigates technical debt, there is a minor risk of "dual-use." Specifically, automated refactoring agents could inadvertently introduce logic flaws, hidden security vulnerabilities (e.g., CWEs), or anti-patterns if deployed in production environments without human supervision. To mitigate this risk, we emphasize that SmellBench is strictly a diagnostic evaluation framework. We strongly advocate that code generated or refactored by LLM-based agents should undergo mandatory human-in-the-loop code review before deployment in mission-critical software systems.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

AI Aider. 2023. Aider: Ai pair programming in your terminal.

Anthropic. 2025. Claude model report. https://www.anthropic.com/transparency/model-report. Accessed 2026-02-09.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and 1 others. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36:46701–46723.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*.

Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2025. Perfbench: Can agents resolve real-world performance bugs? *arXiv preprint arXiv:2509.24091*.

Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam. 2025. Refactorbench: Evaluating stateful reasoning in language agents through code. *arXiv preprint arXiv:2503.07832*.

Harbor Framework Team. 2026. Harbor: A framework for evaluating and optimizing agents and models in container environments.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Ziga Kovacic, Justin T Chiu, Celine Lee, Wenting Zhao, and Kevin Ellis. 2025. Refactoring codebases through library design. *arXiv preprint arXiv:2506.11058*.

Yonnel Chen Kuang Piao, Jean Carlors Paul, Leuson Da Silva, Arghavan Moradi Dakhel, Mohammad Hamdaqa, and Foutse Khomh. 2025. Refactoring with llms: Bridging human expertise and machine understanding. *arXiv e-prints*, pages arXiv–2510.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025a. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Bo Liu, Yanjie Jiang, Yuxia Zhang, Nan Niu, Guangjie Li, and Hui Liu. 2025b. Exploring the potential of general purpose llms in automated software refactoring: an empirical study. *Automated Software Engineering*, 32(1):26.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, and 66 others. 2026. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *Preprint*, arXiv:2601.11868.

Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Wang You, Zhenheng Tang, Yuntao Du, Bill Sun, Hongzhang Liu, and 1 others. 2025. Gittaskbench: A benchmark for code agents solving real-world tasks through code repository leveraging. *arXiv preprint arXiv:2508.18993*.

Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Timofey Bryksin, and Danny Dig. 2024. Next-generation refactoring: Combining llm insights and ide capabilities for extract method. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 275–287. IEEE.

Md Raihan Tapader, Md Mostafizer Rahman, Ariful Islam Shiplu, Md Faizul Ibne Amin, and Yutaka Watanobe. 2025. Code refactoring with llm: A comprehensive evaluation with few-shot settings. *arXiv preprint arXiv:2511.21788*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Yisen Xu, Jinqiu Yang, Tse-Hsun, and Chen. 2026. Swe-refactor: A repository-level benchmark for real-world llm-based code refactoring. *Preprint*, arXiv:2602.03712.

Kaiwen Yan, Hongcheng Guo, Xuanqing Shi, Jingyi Xu, Yaonan Gu, and Zhoujun Li. 2025. Codeif: Benchmarking the instruction-following capabilities of large language models for code generation. *arXiv preprint arXiv:2502.19166*.

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

John Yang, Carlos E 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*, 37:50528–50652.

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. *Preprint*, arXiv:2504.02605.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Datasets

Table 2 summarizes the statistics of the SmellBench benchmark. The benchmark contains 294 cases covering seven widely used open-source Python projects: `click`, `jinja`, `numpy`, `pandas`, `scikit-learn`, `seaborn`, and `xarray`. On average, each case involves 232.5 lines of code changes, 3.8 files, and 4.3 methods, highlighting the complex multi-file and tightly coupled refactoring characteristics of SmellBench. Moreover, each case contains an average of 23.0 test cases to ensure functional correctness of our SmellBench.

Table 2: Statistics of SmellBench.

Project	Lines	Files	Methods	Tests	Count
click	209	3.4	3.8	24.0	42
jinja	199	3.2	4.2	23.1	42
numpy	241	6.9	4.1	11.4	42
pandas	281	3.4	4.9	37.6	42
scikit	221	2.9	4.3	21.0	42
seaborn	193	3.1	4.1	20.5	42
xarray	284	3.6	4.5	23.6	42
Overall	**232.5**	**3.8**	**4.3**	**23.0**	**294**

B Token Consumption

In this section, we discuss the token consumption of our framework. The total token usage consists of two components: generation pipeline consumption and evaluation consumption. As the token cost of the LLM-as-Judge stage is negligible compared to the other components, we omit it from the statistics.

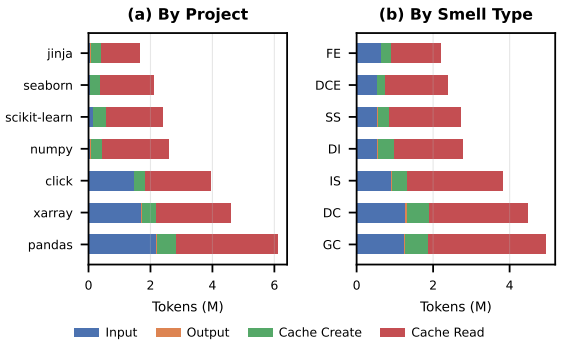

Figure 7: Token consumption in the generation pipeline: (a) grouped by repository; (b) grouped by smell type.

B.1 Token Consumption in the Generation

The generation pipeline introduces token consumption in three stages: Candidate Discovery, Smell Injection, and Fixing Errors. All cases in our experiments were generated using Claude Code with Claude Opus 4.6. Detailed statistics for different repositories and smell types are shown below.

Figure 7 shows that generating one case consumes around 2M tokens on average. Most of these tokens are cached tokens, which helps reduce the actual cost significantly. The click, seaborn, and pandas repositories exhibit the highest token consumption. This is mainly because candidate methods in these repositories contain complex dependencies and strong interconnections, requiring multiple synchronized modifications during code transformation.

Similarly, the data clumps smell type often requires modifications across multiple locations, making the identification of related methods highly expensive in terms of tokens. Furthermore, god classes are usually located in large classes or files, where understanding class functionality and dependencies requires substantial contextual reasoning, leading to higher token consumption.

B.2 Token Consumption in Evaluation

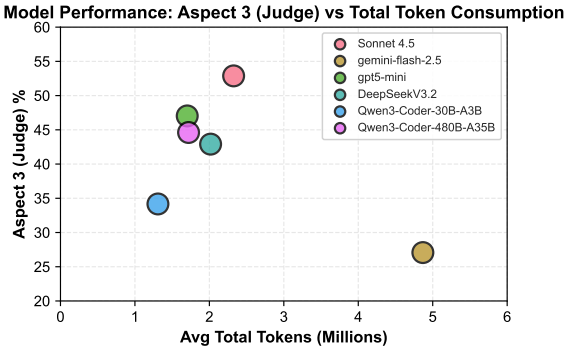

Figure 8: Comparison of model accuracy and efficiency. Higher scores with lower token consumption indicate better efficiency.

In this subsection, we further analyze the relationship between model performance and token consumption in order to investigate which models achieve a better trade-off. Based on OpenHands as the evaluation framework, we evaluated the performance and token consumption of six different LLMs. The results are shown in Figure 8.

The experimental results indicate that Sonnet 4.5 achieves the best overall performance among all evaluated models, while maintaining token consumption within an acceptable range, demonstrating strong overall efficiency. In contrast, the open-source models DeepSeek V3.2 and Qwen3-Coder-480B-A35B exhibit performance and token consumption comparable to gpt-5-mini, suggesting that recent high-performance open-source models are increasingly capable of matching the effectiveness and efficiency of proprietary models.

By comparison, Gemini-Flash-2.5 shows relatively weaker overall capability due to its earlier release. It not only consumes more tokens during evaluation, but also significantly underperforms compared with the other models.

C Smell Types

This appendix section describes the code smell types used to construct refactoring cases in our

benchmark. Each smell represents a recurring structural deficiency that motivates non-trivial refactoring beyond local edits, often involving interactions across functions, classes, or files.

- **Feature Envy**: Methods that rely heavily on data or behavior from other classes, indicating misplaced responsibilities that require redistribution across classes.
- **God Classes**: Classes that accumulate excessive responsibilities, motivating decomposition into smaller, more cohesive components.
- **Data Clumps**: Groups of variables that frequently co-occur across interfaces, suggesting the need for introducing new abstractions.
- **Shotgun Surgery**: Designs where a single change triggers scattered edits across the codebase, indicating poor localization of responsibilities.
- **Dead Code Elimination**: The presence of unreachable or unused code that can be safely removed through whole-program reasoning.
- **Interface Segregation**: Overly broad interfaces that force clients to depend on unused methods, motivating interface decomposition.
- **Deep Inlining**: A refactoring scenario in which deeply nested call chains are collapsed into a single caller, significantly increasing local complexity despite preserving semantics.

D Difficulty Levels

Each refactoring case is further stratified into three difficulty levels (*easy*, *medium*, and *hard*) to control structural complexity and refactoring effort. Difficulty is determined by factors such as the number of files involved, and how strongly the smell is obscured by legitimate-looking design patterns.

As an illustrative example, the difficulty specification for **Feature Envy** is defined as follows:

- **Easy**: Move one method's core logic to access another class's data directly, spanning exactly two files, with explicit foreign attribute access revealing the envy.
- **Medium**: Distribute a method's logic across three or more files via indirect delegation chains (e.g., A calls B which accesses C's internal data), hide the dependency behind wrapper or adapter patterns, and interleave it with legitimate helper methods to introduce diff noise.
- **Hard**: Introduce envy through mixin, decorator, or dynamic dispatch patterns so that the cross-class dependency appears intentional; span four or more files with indirect coupling (e.g., callbacks or shared state), include at least one red herring, and require semantic understanding to identify the smell.

These difficulty specifications are directly incorporated into the prompt used for case generation. Detailed specifications for all smell types and difficulty levels are provided in `smell_types.json` in the https://anonymous.4open.science/r/SmellBench-5602/ .

E Instruction Settings

We define two instruction settings to evaluate different aspects of model capability: *guided* and *targeted*. Instructions are generated at the instance level, meaning that each refactoring case is associated with a distinct natural-language prompt.

- **Guided**: The instruction specifies only the smell type and the affected file, requiring the model to first localize the smell before performing refactoring.
- **Targeted**: The instruction specifies the smell type, file, and the exact class or method involved, focusing the task on executing the refactoring itself.

For example, for case `b49a19838bb`, the *guided* instruction is:

> Feature envy has been detected in `src/click/core.py`. Can you eliminate this smell by moving the logic to a more appropriate location?

The corresponding *targeted* instruction is:

> The `parse_args` method in the `Command` class within `src/click/core.py` exhibits feature envy. Please address this design issue by relocating the behavior to where the data naturally belongs.

All instance-level instructions are available in the SmellBench.

F Rubrics

In this subsection, we present the scoring criteria for the four rubrics. Specifically, we adopt a 10-point scoring system, where a corresponding description is provided for every two-point interval. For better presentation and readability, Table 1 converts these scores into a normalized 1-point scale. The detailed criteria are as follows:

- **Code Quality**. Improves code clarity, style, and overall quality to make the code easier to understand and maintain.

  - **0-1** score: Significantly worse readability than before.

  - **2-3** score: Poor quality; hard to follow.

  - **4-5** score: Below average; multiple readability concerns.

  - **6-7** score: Acceptable quality; some naming or structural issues.

  - **8-9** score: Good quality; minor naming or style improvements possible.

  - **10** score: Clean, idiomatic, well-named, easy to maintain.

- **Structural Soundness**. Maintains a well-organized architecture with clear module boundaries and reliable design principles

  - **0-1** score: Introduces new code smells or anti-patterns.

  - **2-3** score: Significant structural problems.

  - **4-5** score: Noticeable structural issues; responsibilities not well separated.

  - **6-7** score: Reasonable but some unnecessary complexity.

  - **8-9** score: Sound structure with minor imperfections.

  - **10** score: Proper decomposition; single responsibility; appropriate abstraction.

- **Cross-File Coordination**. Ensures consistency and correct interaction across multiple files, modules, and dependencies.

  - **0-1** score: Cross-file coordination largely missing or incorrect.

  - **2-3** score: Minimal cross-file awareness; related code in other files ignored.

  - **4-5** score: Some cross-file changes made but several inconsistencies or leftovers.

  - **6-7** score: Core changes correct; some related artifacts (unused helpers, stale imports) remain in other files.

  - **8-9** score: Nearly all cross-file impacts handled; one minor leftover.

  - **10** score: - 10: All smell-related code properly addressed; no orphaned imports, dead helpers, or dangling references left behind.

- **Smell Elimination**. Removes code smells such as duplication, unnecessary complexity, and fragile patterns to improve maintainability.

  - **0-1** score: Smell not addressed or new smell introduced.

  - **2-3** score: Minimal effort; smell barely addressed.

  - **4-5** score: Only the most obvious smell location fixed; secondary artifacts untouched.

  - **6-7** score: Core smell addressed but some related artifacts (unused helpers, stale registrations) left behind.

  - **8-9** score: Smell substantially eliminated; only minor traces remain.

  - **10** score: Smell fully eliminated; no residual smell code, unused imports, or orphaned helpers remain.

G Case Study

This section presents a concrete case study to illustrate how the same smell type differs across difficulty levels. We use the dead code elimination task from the Click project as an example. As shown in Figure 9, the easy-level task only involves modifications to two files and introduces one new function, `_check_command_alias_conflict`, where the dead code appears as an unreachable conditional branch `resolved_name is not None`. Figure 10 demonstrates that the medium-level task involves three files, with significantly higher complexity than the easy level. Figure 11 shows that the hard-level task involves four files and introduces the decorator `_command_resolver`, further increasing the concealment of the dead code.

H Prompt Design

Below are the prompt templates in our pipelines.

Case Study: Easy Dead Code in Click

1. Smell Specification

```
"instance_id": ""click-dead_code_elimination-e14f8e46a5034178a0872e65de70d4c9",
"smell_function": [src/click/core.py, Group , resolve_command],
"difficulty": "easy",
"type": "dead_code",
...
```

2. Calling Chain Overview

File: src/click/core.py

```
...
if cmd is not None and cmd_name != original_cmd_name:
    alt_cmd = self.get_command(ctx, original_cmd_name)
    if alt_cmd is not None and alt_cmd is not cmd:
        resolved_name = _check_command_alias_conflict(
            cmd_name, original_cmd_name, self.list_commands(ctx)
        )
        if resolved_name is not None:
            cmd_name = resolved_name
            cmd = self.get_command(ctx, resolved_name)

...
```

File: src/click/utils.py

```
def _check_command_alias_conflict(
    normalized_name: str,
    original_name: str,
    available_commands: cabc.Sequence[str],
):
    posix_normalized = _posixify(normalized_name)
    posix_original = _posixify(original_name)
    if posix_normalized == posix_original:
        return make_str(normalized_name)
    for listed_name in available_commands:
        listed_posix = _posixfy(listed_name)
        if listed_posix == posix_normalized:
            return normalized_name
        if listed_posix == posix_original:
            return original_name

    return None
```

Figure 9: Easy Dead Code Elimination in Click

Case Study: Medium Dead Code in Click

1. Smell Specification

```
"instance_id": "click-dead_code_elimination-7a8e2a494a7aea9c10ale1d4f3990546",
"smell_function": [src/click/shell_completion.py, null , resolve_context],
"difficulty": "medium",
"type": "dead_code",
...
```

2. Calling Chain Overview

File: src/click/shell_completion.py

```
...
if resolution_mode is None:
    resolution = _RESOLUTION_MODE
...
policy: _ContextResolutionPolicy | None = None
if resolution_mode is not ResolutionMode.STANDARD:
    policy = _ContextResolutionPolicy(resolution_mode)
...
if policy is not None:
    policy.enter_content(ctx)
...
return _validator_context_resolution(ctx, policy, args)
```

File: src/click/_utils.py

```
class ResolutionMode(enum.Enum):

    STANDARD = "standard"
    VALIDATED = "validated"
    STRICT = "strict"
```

File: src/click/core.py

```
class _ContextResolutionPolicy:
    ...
    def enter_context(self, ctx: Context):
        ...
    def validate_resolved(self, ctx: Context):
        ...
        return ctx
```

File: src/click/core.py

```
def _validate_context_resolution(
    ctx: Context,
    policy: _ContextResolutionPolicy | None,
    args: list[str],
):
    if policy is None:
        return ctx
    if not policy.requires_validation:
        return ctx
    return policy.validate_resolved(ctx)
```

Figure 10: Medium Dead Code Elimination in Click

Case Study: Hard Dead Code in Click

1. Smell Specification

```
"instance_id": "click-dead_code_elimination-721e7057ad4d0e4.aa420e921a264e9f1",
"smell_function": [src/click/core.py, Group, resolve_command],
"difficulty": "hard",
"type": "dead_code",
...
```

2. Calling Chain Overview

File: src/click/core.py

```
resolvers = _command_resolver.get_resolvers(_command_resolver.active_mode)
...
if resolvers:
    metadata = get_resolution_metadata()
    for resolver in resolvers:
        resolved_name, resolved_cmd = resolver(
            cmd_name, self, ctx, metadata
        )
        if resolved_cmd is not None:
            return resolved_name, resolved_cmd, args[1:]
...
```

File: src/click/_utils.py

```
class _ResolutionStrategy:
    ...
    def get_resolvers(self, mode: str):
        entries = self._strategies.get(mode, [])
        return [fn for _, fn in entries]
    ...
```

File: src/click/globals.py

```
def get_resolution_metadata():
    ctx = get_current_context(silent=True)
    metadata = {"has_context": ctx is not None}
    if ctx is not None:
        ...
    return metadata
```

File: src/click/util.py

```
@_command_resolver.register("extended", priority=10)
def _extended_prefix_resolver(
    cmd_name: str,
    ...
):
    ...
    return None, None
```

```
@_command_resolver.register("standard", priority=10)
def _standard_passthrough_resolver(
    cmd_name: str,
    ...
):
    ...
    return None, None
```

Figure 11: Hard Dead Code Elimination in Click

Prompt for Candidate Discovery

You are a code analysis expert. Your task is to find methods/classes where it would be **convenient to inject** a specific code smell — NOT to find places that already exhibit this smell.
The goal is to identify code locations where the structure, complexity, and cross-module relationships make it natural and easy to introduce the smell while keeping the code compilable and tests passing.

Source path: ‘[SRC_PATH]‘

Eligible files: [ELIGIBLE_FILES]

Smell type to inject: [SMELL_SPECIFICATION]

Strategy
Do NOT read every file. Instead:
1. Based on file names and module structure, pick the 3-5 most promising files
2. Use ‘grep‘ to quickly locate class definitions, large methods, and cross-module interactions
3. Only read specific sections of files (use line ranges) to verify candidates
4. Prioritize files with core business logic (e.g., core.py, models.py, engine.py) over peripherals

Requirements
Find exactly 5 candidates. Each candidate should be a method/class where injecting ‘[SMELL_TYPE]‘ would be **easy and natural** — meaning the surrounding code structure supports the injection without breaking functionality.

Each candidate needs:
- ‘file‘: relative path from repo root
- ‘class_name‘: class name (null if standalone function)
- ‘method_name‘: method/function name (for god_classes/interface_segregation, can be null)
- ‘line_number‘: the actual starting line number (verify by reading)
- ‘reason‘: 1-2 sentences explaining why this location is a good **injection point** (what structural properties make it easy to introduce the smell here)

IMPORTANT: At most 2 candidates may come from the same file. Spread candidates across different files to ensure diversity.

DIVERSITY REQUIREMENT: The 5 candidates must be substantially different from each other:
- They should involve different classes/functions with different responsibilities
- They should target different code patterns or architectural concerns
- Avoid picking multiple methods from the same class or methods that do similar things

Output After finding all candidates, output a single JSON block:

```
{
  "[SMELL_TYPE]": [
    {"file": "...", "class_name": "...", "method_name": "...", "line_number":
        123, "reason": "..."}
  ]
}
```

The key must be exactly: [SMELL_TYPE]
It must have exactly 5 entries.

Prompt for Case Generation

You are a senior software engineer specializing in code transformation and refactoring benchmarks. Your task is NOT to improve the code. Instead, you must inject a specific type of code smell into the given codebase.

Objective
Inject the following code smell: [SMELL_TYPE]
into the target location: [PROJECT_PATH]

Target Candidate
You MUST inject the smell starting from the following candidate: [CANDIDATE PATH]

Difficulty Level: [DIFFICULTY] [DIFFICULTY_DESC]

Constraints
[Constraints Details]

Output Requirements
After generating the modified codebase, you MUST append a JSON object at the very end of your response. Format:

```
{
  "smell_type": "Type name",
  "hint_targeted": "...",
  "hint_guided": "...",
  "smell_function": ["absolute/path/to/file.java", "ClassName", "methodName"],
  "test_functions": [
    ["absolute/path/to/file.java", "ClassName", "methodName"],
    ["absolute/path/to/other.java", "ClassName", "methodName"]
  ]
}
```

smell_function rules
[Smell Function DESC]

test_functions rules
[TEST FUNCTION DESC]

Prompt for Fixing Errors

You previously injected a "[SMELL_TYPE]" code smell into this codebase.
The changes you made caused the following unit tests to FAIL.

CRITICAL: Your task is to fix the TESTS, NOT to remove the smell!

Your task: Fix the code so that all tests pass while
KEEPING the injected code smell intact.
- The smell must still be present and non-trivial after your fix
- You are fixing **test failures caused by your injection**, NOT refactoring the smell away
- Think of it as: "make my smell injection more robust so it doesn't break tests"

DO NOT:
- Remove or refactor the smell you injected
- "Clean up" the code smell
- Make the code better quality — keep the smell as is

DO:
- Fix any syntax errors, import errors, or runtime errors
- Adjust test-related code if needed
- Make minimal changes to pass tests while preserving the smell pattern

Do NOT run any tests yourself — testing is handled externally.

Diff of your previous changes
[SMELL_CONTENT]

Failing test scripts
[TEST_SCRIPTS]

Test error output
[TEST_ERROR_OUTPUT]

Requirements
1. **Fix the failing tests while preserving the code smell injection** — the smell pattern must remain
2. The code must compile/run correctly
3. **DO NOT remove or refactor the smell** — you are fixing test failures, not improving code quality
4. DO NOT create new files
5. DO NOT run any test commands (pytest, unittest, etc.)
6. Make **minimal changes** — only fix what's broken, keep the smell as you originally injected it

After making your fixes, output the same JSON format as before:

```
{
  "smell_type": "Type name",
  "hint_targeted": "...",
  "hint_guided": "...",
  "smell_function": ["absolute/path/to/file", "ClassName", "methodName"],
  "test_functions": [
    ["absolute/path/to/file", "ClassName", "methodName"],
    ["absolute/path/to/other", "ClassName", "methodName"]
  ]
}
```

Prompt for Smell Analysis

You are an expert software engineer analyzing a code change (diff) that introduces a "smell_type" code smell into a codebase.

Smell Type
- **Name**: smell_type
- **Description**: smell_description

The Smell Diff
This diff was applied to a clean codebase to introduce the smell:
[smell_diff]

Your Task
Analyze this diff by going through each change and assessing its significance. Focus on **what each change means and how important it is**, NOT on how to fix it.
For each distinct change in the diff (a new function, a moved block, an added import, etc.), explain:
- **What it does**
- **How significant it is** (critical / moderate / minor) to the smell
- **What it degrades** in the codebase (e.g., coupling, cohesion, readability, API surface, etc.)

After covering individual changes, provide:
1. **Overall smell pattern**: Summarize how these changes work together to create the "smell_type" smell. What design principle is violated?
2. **Severity ranking**: Rank the changes from most to least important. Which changes are the **root cause** of the smell, and which are just supporting noise?
3. **What was degraded overall**: What concrete qualities of the codebase were harmed? Be specific about the impact on maintainability, coupling, cohesion, etc.
4. **Key evaluation signals**: When judging whether a candidate fix truly addresses this smell, what should matter most? What would distinguish a thorough fix from a superficial one?

Output Format
Return your result using XML tags. Do NOT wrap the output in a code block.
<analysis>
Your full analysis text as described above. Write freely — no escaping needed.
</analysis>

Example Output for Smell Analysis

Change 1: Storing ‘_last_state‘ in ‘_OptionParser‘ (parser.py)
What it does: Adds a single line ‘self._last_state = state‘ to persist the parsing state object after parsing completes. This exposes internal parsing state that was previously encapsulated within the parse method.

Significance: CRITICAL - This is the enabling infrastructure change that makes the feature envy possible. Without this, the Command class couldn’t access the parser’s internal state.

What it degrades:
- **Encapsulation**: Breaks the parser’s encapsulation by exposing internal state (‘_ParsingState‘) that should remain private - **API surface**: Implicitly expands the parser’s interface by making ‘_last_state‘ a queryable attribute

- **Coupling**: Creates a temporal coupling - the ‘_last_state‘ is only valid after parsing, but there’s no contract enforcing this
...

Figure 12: Example output of smell analysis used in LLM-as-Judge.

Prompt for Task Instruction

You are an expert Python refactoring assistant.
Task
[guided or targeted instruction]

Instructions
1. Read and understand the relevant code in the project.
2. Identify the code smell and understand why it is problematic.
3. Refactor the code to eliminate the smell while preserving all existing behavior.
4. Ensure all tests continue to pass after your changes.

Constraints
- Do not change the original program behavior.
- Preserve the original control flow, inputs, and outputs.

IMPORTANT
You are allowed to perform any operations, such as checking or modifying related files, without prior approval from me.

Response

Figure 13: Task instruction prompt provided to the refactoring agent.

Prompt for LLM-as-Judge

You are an expert code reviewer evaluating a refactored version of code that originally contained a "[smell_type]" code smell.

Context
- **Smell Type**: [smell_type]
- **Smell Description**: [description]
[smell_analysis]

IMPORTANT: The refactoring below is provided in **unified diff format** (git diff output). Lines starting with '-' are removed, lines starting with '+' are added, and context lines are unchanged. Evaluate the *intent and quality of the changes*, not the completeness of the code shown — diffs only show changed regions, not the full files.

Refactoring (diff fixing the smell)
[refactored_code]

Evaluation Rubric
[general_dimensions]

Return your evaluation as JSON:

```
{
  "smell_elimination": {
    "score": 0,
    "justification": "brief explanation"
  },
  "cross_file_coordination": {
    "score": 0,
    "justification": "brief explanation"
  },
  "structural_soundness": {
    "score": 0,
    "justification": "brief explanation"
  },
  "code_quality": {
    "score": 0,
    "justification": "brief explanation"
  },
  "summary": "2-3 sentence overall assessment"
}
```

Figure 14: Judging Prompt for LLM to evaluate the refactoring result of agents.