FEATBENCH: EVALUATING CODING AGENTS ON FEATURE IMPLEMENTATION FOR VIBE CODING

Anonymous authors
Paper under double-blind review

ABSTRACT

The rapid advancement of Large Language Models (LLMs) has given rise to a novel software development paradigm known as "vibe coding," where users interact with coding agents through high-level natural language. However, existing evaluation benchmarks for code generation inadequately assess an agent's vibe coding capabilities. Existing benchmarks are misaligned, as they either require code-level specifications or focus narrowly on issue-solving, neglecting the critical scenario of feature implementation within the vibe coding paradiam. To address this gap, we propose FeatBench, a novel benchmark for vibe coding that focuses on feature implementation. Our benchmark is distinguished by several key features: ❶ **Pure Natural Language Prompts.** Task inputs consist solely of abstract natural language descriptions, devoid of any code or structural hints. ❷ **A Rigorous & Evolving Data Collection Process.** FeatBench is built on a multi-level filtering pipeline to ensure quality and a fully automated pipeline to evolve the benchmark, mitigating data contamination. ❸ **Comprehensive Test Cases.** Each task includes Fail-to-Pass (F2P) and Pass-to-Pass (P2P) tests to verify correctness and prevent regressions. ❹ **Diverse Application Domains.** The benchmark includes repositories from diverse domains to ensure it reflects real-world scenarios. We evaluate two state-of-the-art agent frameworks with four leading LLMs on FeatBench. Our evaluation reveals that feature implementation within the vibe coding paradigm is a significant challenge, with the highest success rate of only 29.94%. Our analysis also reveals a tendency for "aggressive implementation," a strategy that paradoxically leads to both critical failures and superior software design. We release FeatBench, our automated collection pipeline, and all experimental results to facilitate further community research. Our code is available at `https://anonymous.4open.science/r/FeatBench-D3C5`.

"The hottest new programming language is English."

—Andrej Karpathy (Karpathy, 2025b)

1 INTRODUCTION

The rapid advancement of Large Language Models (LLMs) has given rise to a novel software development paradigm(Jiang et al., 2024; Li et al., 2024; Seed et al., 2025), recently termed "Vibe Coding."(Horvat, 2025; Karpathy, 2025a; Wikipedia, 2025) This approach allows users to program by interacting with an LLM-powered coding agent through high-level, abstract requests in natural language. The agent then autonomously generates, tests, and executes the code, obviating the need for users to write or review it. In this paradigm, authorship transfers to the AI without needing to understand the implementation details. Vibe coding is transformative for implementing novel ideas, particularly for users unfamiliar with programming. For instance, a product manager can describe a new feature to generate a prototype. In another scenario, a data analyst employs rapid, iterative coding for Exploratory Data Analysis (EDA). These examples illustrate that vibe coding can enhance productivity and foster discoveries for practitioners across diverse fields. Consequently, vibe coding is emerging as a prominent topic in artificial intelligence, drawing significant research interest.

High-quality evaluation benchmarks are essential to foster the development of vibe coding. However, existing evaluation benchmarks for code generation inadequately assess an agent's vibe coding capabilities. Traditional code generation benchmarks, such as HumanEval(Chen et al., 2021) and

Table 1: Comparison with existing Vibe Coding benchmarks.

Benchmark	Date	Task Type	Curation	Level	Task input
SWE-bench(Jimenez et al., 2024)	Oct, 2023	Issue Solving	Manual	Repository Level	Issue Description
SWE-bench-Live(Zhang et al., 2025)	Jun, 2025	Issue Solving	Automatic	Repository Level	Issue Description
RACodeBench(Zhao et al., 2025)	Sep, 2025	Issue Solving	Manual	Repository Level	Issue Description+Wrong Code
FeatBench (Ours)	Sep, 2025	**Feature Implementation**	**Automatic**	**Repository Level**	**Feature Requirement**

ClassEval(Du et al., 2023), are misaligned with the vibe coding paradigm. Beyond natural language requirements, their inputs demand code-level specifics like function signatures, a methodology fundamentally misaligned with the vibe coding paradigm. The benchmarks most relevant to the vibe coding paradigm are issue-solving benchmarks, represented by SWE-Bench(Jimenez et al., 2024), which generate code patches based solely on issue descriptions. However, these benchmarks cover only the issue-solving scenario, neglecting other critical scenarios in vibe coding, such as feature implementation, as demonstrated in our comparison in Table 1. Therefore, it is necessary to construct more diverse benchmarks to evaluate agents' vibe coding capabilities comprehensively.

We propose FeatBench, a novel benchmark for vibe coding to fill this research gap. Unlike issue-solving benchmarks (e.g., SWE-bench), our benchmark focuses on a critical yet under-evaluated aspect of vibe coding: feature implementation. This task involves implementing new functionalities based on abstract natural descriptions from a user's perspective, directly simulating how non-technical users add capabilities to existing software. It is an everyday real-world development activity, supported by existing studies(LaToza et al., 2006) indicating that developers spend approximately 37% of their time on new feature development. *To the best of our knowledge, FeatBench is the first benchmark to evaluate an agent's proficiency in feature implementation within the vibe coding paradigm.* Our benchmark is distinguished by several key features: ❶ **Pure Natural Language Prompts:** Task inputs consist solely of abstract natural language descriptions, devoid of any code or structural hints, to accurately reflect the vibe coding workflow. ❷ **Rigorous & Evolving Data Collection Process:** Our data collection process follows strict quality assurance standards, with a multi-level filtering pipeline. To mitigate data contamination, we develop a fully automated pipeline to evolve our benchmark without human effort. The initial release of our benchmark comprises 157 tasks sourced from 27 actively maintained open-source GitHub repositories. ❸ **Comprehensive Test Cases:** Each task is equipped with the Fail-to-Pass (F2P) and Pass-to-Pass (P2P) test cases, verifying both the generated code's correctness and the preservation of existing functionality. ❹ **Diverse Application Domains:** To ensure FeatBench accurately reflects real-world development scenarios, it includes repositories from diverse domains such as AI/ML, DevOps, and Web development.

To demonstrate the utility of FeatBench, we evaluate two SOTA agent frameworks, Trae-agent and Agentless, using four SOTA LLMs, including open-source models like DeepSeek V3.1(Deepseek, 2025) and proprietary models such as GPT-5(OpenAI, 2025), Doubao-Seed-1.6(ByteDance, 2025), and Qwen3-Coder-Flash(Team, 2025). Following established standards(Deng et al., 2025; Zhang et al., 2025), we adopt the **Resolved Rate (%)** as our primary metric. We also report the **Patch Apply Rate (%)** and the **File-level Localization Success Rate (%)**, alongside several auxiliary metrics. Based on our experimental results, we find that: ❶ FeatBench poses a significant challenge to SOTA agents, with the top-performing configuration achieving a resolved rate of only 29.94%. We identify an apparent performance disparity between agent paradigms: autonomous, planning-based agents substantially outperform rigid, pipeline-based counterparts. ❷ A critical and widespread failure mode is the introduction of regressions. All evaluated agents tend to break existing functionalities when adding new features. This undermines the reliability required for the vibe coding paradigm, where the user does not typically review code. ❸ Our case studies found that agents often adopt an "aggressive implementation" strategy when adding new features. This behavior acts as a double-edged sword. While this strategy is the primary cause of task failures through "scope creep," it can also yield solutions with superior software architecture and robustness. This finding highlights the critical need for mechanisms to control this behavior.

2 FeatBench

2.1 Task Definition

Figure 1 shows a sample in FeatBench. Each sample consists of four components: ❶ **Feature Description:** A detailed natural language description of the new feature to be added. ❷ **Running**

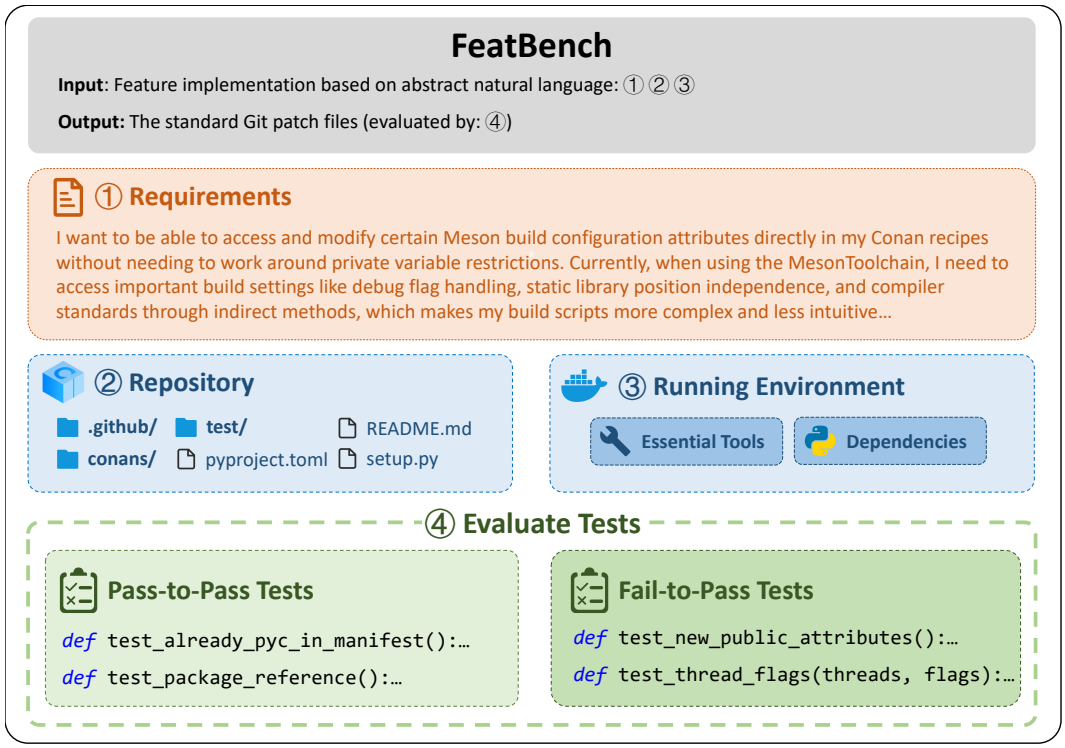


Figure 1: An overview of FeatBench. Each sample consists of four components.

Environment: A pre-configured Docker image containing the runtime environment. ❸ **Repository:** The complete codebase at the required commit. ❹ **Evaluation Tests:** A comprehensive suite of Pass-to-Pass (P2P) tests to detect regressions and Fail-to-Pass (F2P) tests to verify the correct implementation of the new feature.

2.2 Features of FeatBench

Pure Natural Language Prompts. To precisely simulate a user's intent, we mandate that all prompts are framed as a first-person feature request, beginning with "I want to...". These prompts, averaging 1848 characters, are stripped of implementation details and structured to include functional appeal, background motivation, and specific requirements. This design ensures the agent receives a comprehensive and unambiguous task description.

Rigorous & Evolving Data Collection Process. Our data collection process is designed to be both rigorous and evolving. We employ a strict, multi-level filtering process at the repository, release, PR, and test case levels to ensure rigor. To mitigate data contamination, we develop and open-source a fully automated pipeline to evolve the benchmark. This paper releases the benchmark's initial version, comprising 157 tasks from 27 repositories, with all constituent releases from the past year. We plan to leverage this pipeline to update the benchmark every six months. This process is highly cost-effective, with a processing cost of approximately $0.28 per sample using the DeepSeek V3.1 model. Detailed task statistics are provided in Section A.4.

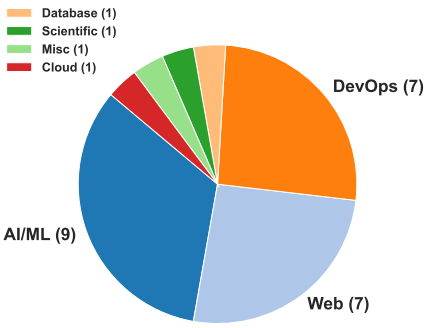


Figure 2: Repository distribution.

Comprehensive Test Cases. The correctness of each generated implementation is rigorously validated. On average, each task is equipped with 1657.53 test cases. This extensive test coverage not only validates the correctness of new features but also robustly detects potential regressions.

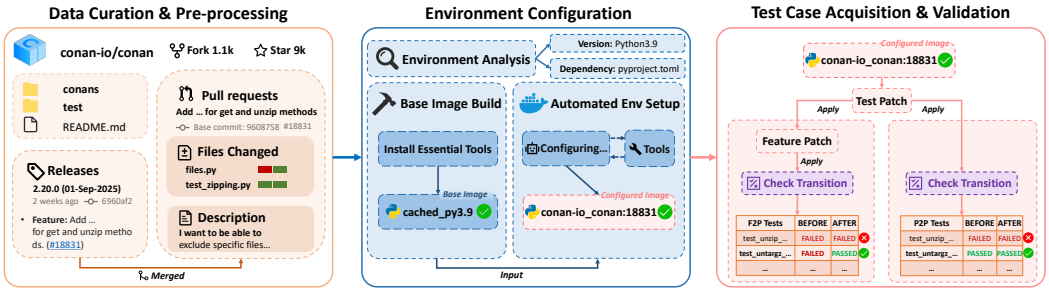


Figure 3: The pipeline of building our benchmark.

Diverse Application Domains. Our benchmark spans a wide range of application domains, with a detailed distribution shown in Fig. 2. This diversity provides a broad platform for cross-domain evaluation of LLM performance. We plan to continually expand our dataset with samples from cutting-edge open-source repositories to maintain the benchmark's challenge and generality.

2.3 Benchmark Construction Pipeline

Our benchmark construction pipeline is divided into three main phases: data curation, environment configuration, and test case validation, as illustrated in Fig. 3. Further technical details are provided in the Section A.2.

Data Curation and Pre-processing. The initial filtering occurs at the repository level, where we selected 27 high-quality candidates from an initial pool of 44 open-source repositories based on several criteria: test file exists, a history of active maintenance with at least three official releases, and relevance to common software development domains (e.g., excluding tutorials).

We curated 675 releases from these repositories and then analyzed them individually, employing an LLM to identify feature-implementation PRs from the natural language of release notes. This step filters for releases created within the last year, longer than 30 characters, and that are not automated or trivial updates generated by bots. The prompt template is in Section A.8.

The final curation is applied at the PR level. From the filtered releases, we extracted an average of 3 feature-implementation PRs per release, yielding 297 high-quality samples. We then filter for these samples that contain changes to Python files and an accompanying test patch to ensure each task is verifiable and represents a functional code change. Furthermore, we impose a critical constraint: the code patch implementing the feature must only modify existing functions, without adding or deleting any. This constraint is crucial for testability, enabling static, developer-written test cases to validate the agent's implementation reliably. We argue this criterion is unbiased, as we posit that an LLM agent should adhere to the same constraint if a human developer refrains from adding or deleting functions due to complexity or maintainability concerns. The final validation step, ensuring each sample had at least one F2P test case, resulted in 157 tasks. Finally, since real-world PR descriptions are often terse and developer-centric, we use an LLM to synthesize a more comprehensive, code-agnostic user request suitable for a vibe coding prompt. The prompt template is in the Section A.8.

Environment Configuration. We develop an agent to automatically configure the environment by reverting the repository to its state before the PR, analyzing it to infer the Python version and configuration files, and then constructing a Docker image that accurately replicates the historical runtime environment. The agent's task is considered complete only after ensuring the PR's test suite is executable and installing all dependencies, including optional ones.

Test Case Acquisition and Validation. This final phase establishes a robust ground truth for evaluation. Fail-to-Pass (F2P) test cases are extracted from the new or modified tests introduced in the current PR, which serve as direct evidence of successful feature implementation. We validate that these tests transition from failing to passing after applying the feature patch. We identify Pass-to-Pass (P2P) test cases to prevent regressions by running the entire test suite on the pre-patch repository and confirming they still pass after the patch is applied.

Table 2: Performance comparison across different agents & LLMs on FeatBench.

Model	Trae-agent						Agentless					
	Resolved%	Applied%	RT%	FV%	File%	#Token	Resolved%	Applied%	RT%	FV%	File%	#Token
Open-Source Model												
DeepSeek V3.1	22.29%	**100.00%**	42.68%	46.50%	79.11%	2.21M	9.55%	70.70%	32.48%	19.11%	42.28%	0.05M
Closed-Source Model												
Doubao-Seed-1.6	15.92%	**100.00%**	41.40%	26.75%	65.77%	1.07M	10.19%	91.72%	26.75%	19.11%	49.30%	0.08M
Qwen3-Coder-Flash	20.38%	**100.00%**	**50.32%**	37.58%	74.37%	1.72M	7.00%	71.34%	24.84%	14.01%	36.49%	0.05M
GPT-5	**29.94%**	**100.00%**	**50.32%**	**56.05%**	**86.43%**	2.90M	**16.56%**	**98.09%**	**35.67%**	**34.39%**	**67.54%**	0.07M
Average	22.13%	100.00%	46.18%	41.72%	76.42%	1.98M	10.83%	82.96%	29.94%	21.66%	48.90%	0.06M

3 EXPERIMENTS

3.1 SETUP

Agent Selection. To evaluate the feature implementation capabilities of SOTA agents in vibe coding scenarios, we selected two leading frameworks from software engineering and tested them. We chose agent-based evaluation because vibe coding in a no-code context is a complex task that requires not only code generation but also the ability to localize relevant files within a large repository. The selected frameworks are: **Agentless**(Xia et al., 2024), which employs a two-stage pipeline, and **Trae-agent**(Team et al., 2025), which utilizes an autonomous planning solution. Their differing paradigms allow us to gain deeper insights into the capabilities required for effective vibe coding. For Trae-agent, we imposed a maximum limit of 150 steps per task and equipped it with supplementary tools, including `ckg` and `json_edit_tool`. For Agentless, we generally followed the pipeline and settings described in the original paper, which divides the workflow into two primary stages: feature localization and patch generation. Consistent with SWE-bench-Live's(Zhang et al., 2025) evaluation methodology, we omit the reranking stage based on regression testing. Since supporting this step requires substantial infrastructure adaptation beyond our scope, our Agentless evaluation produces a single candidate solution.

Model Selection. We evaluated the performance of these agents using four recent SOTA LLMs, encompassing both proprietary and open-source models: the open-source DeepSeek V3.1 (deepseek-chat), and three proprietary models, GPT-5 (gpt-5-2025-08-07), Doubao-Seed-1.6 (doubao-seed-1-6-250615), and Qwen3-Coder-Flash (qwen3-coder-flash-2025-07-28). For a comprehensive description of the experimental settings and hyperparameters, refer to the Section A.3.

3.2 EVALUATION METRICS

Following the standards set by previous works(Jimenez et al., 2024; Deng et al., 2025), we adopt the **Resolved Rate (%)** as our primary metric. This measures the percentage of vibe coding tasks successfully completed by an agent. We also report the **Patch Apply Rate (%)**, which indicates the proportion of generated patches that are syntactically correct and can be applied to the repository without errors. Furthermore, we measure the **File-level Localization Success Rate (%)**, which assesses whether the set of files modified by the generated patch matches the ground-truth patches. In addition, we introduce three auxiliary metrics to facilitate a more in-depth analysis of test case outcomes:

- **Feature Validation Pass Rate (%):** This metric evaluates the functional completeness of the implemented feature by measuring the pass rate of the F2P test cases.
- **Regression Tests Pass Rate (%):** This metric measures the proportion of tasks where all original functionalities remain intact after applying the generated patch, evaluated by the pass rate of the P2P test cases.
- **Tokens Cost:** The average number of tokens consumed per task.

3.3 PERFORMANCE EVALUATION ON FEATBENCH

We report the performance of all agent-model combinations on FeatBench in Table 2. The results highlight the challenging nature of FeatBench: **even SOTA agent and LLM combinations achieve**

a low Resolved%, peaking at just 29.94%. For comparison, the Trae-agent framework paired with a top model like Claude 3.7 Sonnet(Anthropic, 2025) reached a 65.67% resolution rate on SWE-bench. This stark performance drop on FeatBench demonstrates that benchmarks designed from a professional developer's perspective cannot adequately evaluate an agent's vibe coding capabilities. A comparison of the two agent paradigms shows that **Trae-agent generally outperforms Agentless**. This advantage likely stems from its ability to autonomously plan execution paths, invoke external tools like code graph analysis, and design its own test cases to verify functionality and facilitate feedback-driven optimization. In contrast, Agentless, designed for bug fixing, follows a rigid two-stage "locate-and-patch" pipeline. We argue that this fixed, template-dependent process limits its problem-solving flexibility and lacks reflective or debugging capabilities. Nevertheless, the **significantly lower token consumption of Agentless** suggests it may be more cost-effective for simpler tasks.

A deeper analysis of the metrics reveals a clear divergence in the core capabilities of the two frameworks. Firstly, regarding File%, **Trae-agent's average of 76.42% far surpasses Agentless's 48.90%**. With its dynamic planning and tool interaction capabilities, Trae-agent can more accurately identify the target files for modification within large repositories. Since precise file localization is a critical prerequisite for task success, the deficiency of Agentless in this initial stage is a key bottleneck limiting its overall performance.

Secondly, a significant gap is also evident in the quality of the generated code patches. While all patches generated by Trae-agent are syntactically valid (**100% Applied%**), their functional correctness is limited. The average FV% and RT% are only **41.72% and 46.18%, respectively**, indicating shortcomings in both implementing new features correctly and maintaining existing functionality. In comparison, Agentless performs even more poorly, with its **average FV% (21.66%) and RT% (29.94%) being substantially lower** than those of Trae-agent. This contrast demonstrates that agents with autonomous planning capabilities exhibit greater flexibility and effectiveness in solving complex problems than their pipeline-based counterparts.

Finally, the **universally low RT%** across all model combinations is a noteworthy and widespread observation. This reveals a significant risk of models introducing regressions by breaking existing functionalities while attempting to add new ones. This finding imposes stricter reliability requirements, a critical factor for the vibe coding paradigm, where users delegate implementation details to the agent. Future research must therefore prioritize strategies that ensure repository stability, as this trust is a foundational prerequisite for adopting vibe coding in real-world software engineering.

3.4 Impact of Task Complexity on Resolved Rate

The agent's success, measured by the resolved rate, is significantly influenced by complexity at both the repository and patch levels.

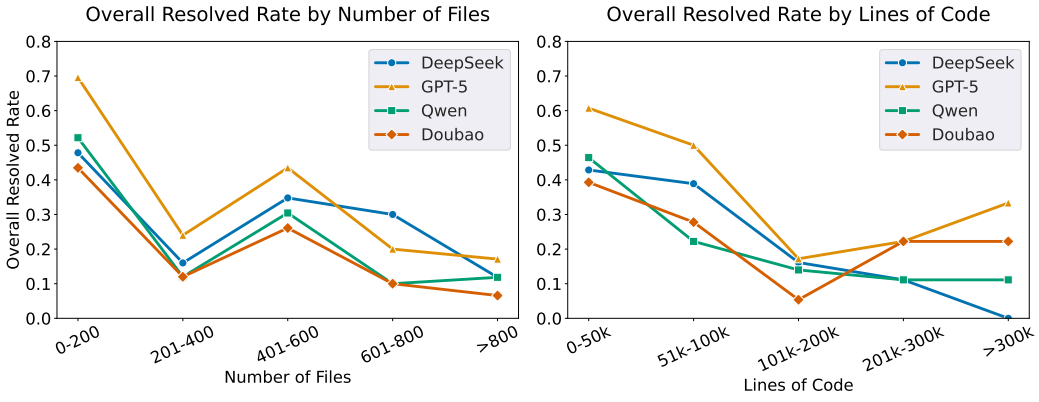


Figure 4: Resolved Rate in Relation to Repository Complexity

At the repository level, complexity is quantified by the total number of files and aggregate lines of code (LOC). As illustrated in Fig. 4, a strong inverse correlation exists between a repository's scale and the agent's performance. Model effectiveness is inversely correlated with repository com-

plexity, as measured by both file count and LOC. While agents perform well on smaller projects (fewer than 200 files or 50,000 LOC), with resolved rates reaching up to 60- 70% for GPT-5, their success degrades sharply with scale. In large repositories (more than 800 files or 300,000 LOC), performance for all models converges to a low of 10–30%. This consistent trend, which affects even top-performing models, highlights that **the ability of the LLM agent systematically declines as project complexity increases, making complexity a fundamental barrier to the success of current agent-based approaches.**

Beyond the overall project, the intrinsic complexity of the ground truth solution, or "golden patch," is another critical factor. As depicted in Fig. 5, which evaluates patch complexity based on its LOC and the number of files it spans, success rates peak for single-file patches and those between 1–30 Lines of Code (LOC), where the top model achieves a 36% resolved rate. However, performance collapses for substantial modifications, with success rates falling to **nearly zero** for patches exceeding 50 LOC or those distributed across five or more files. This highlights a clear limitation in handling larger or more widespread code changes, demonstrating that **the agent's performance is highest on small, localized code changes and degrades significantly as the patch size and distribution increase.**

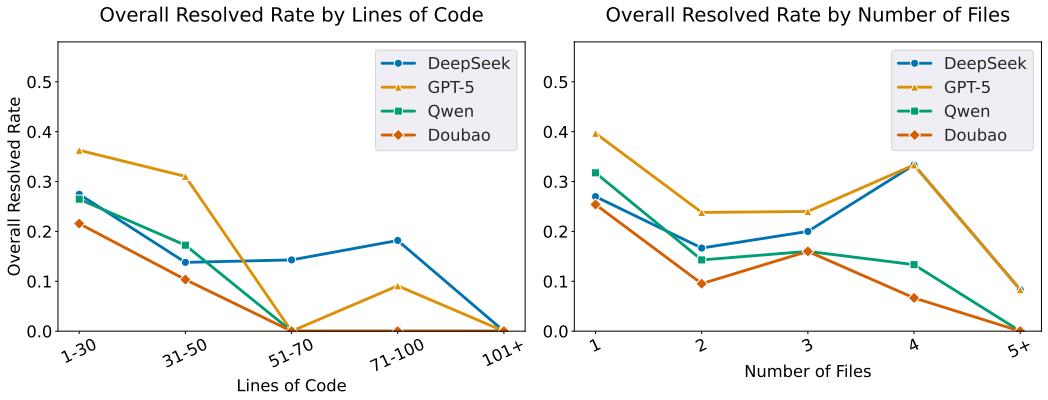


Figure 5: Correlation Between Resolved Rate and Patch Complexity

3.5 Association Between Resolved Rate and Creation Time

An analysis of task creation time confirms **the temporal stability of our benchmark and the absence of data contamination**.

As illustrated in Fig. 6, while the total volume of patches fluctuates across five distinct periods (from 2308 to 2509), the resolved rate for the Trae-agent with Doubao-Seed-1.6 remains remarkably consistent. This stability is critical, as an upward performance trend would suggest data leakage from earlier tasks being repeated in later periods.

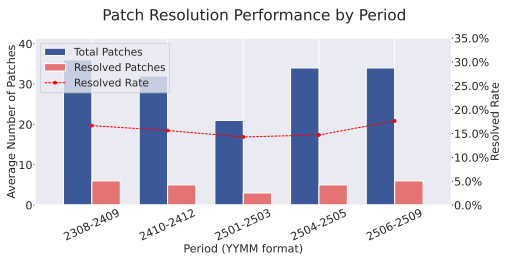


Figure 6: Association Between Resolved Rate and Creation Time

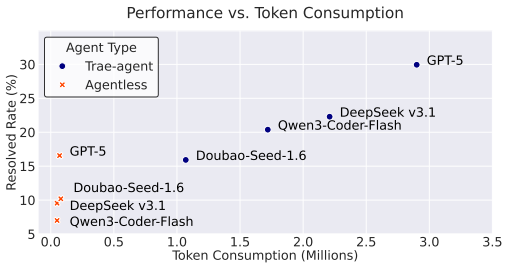
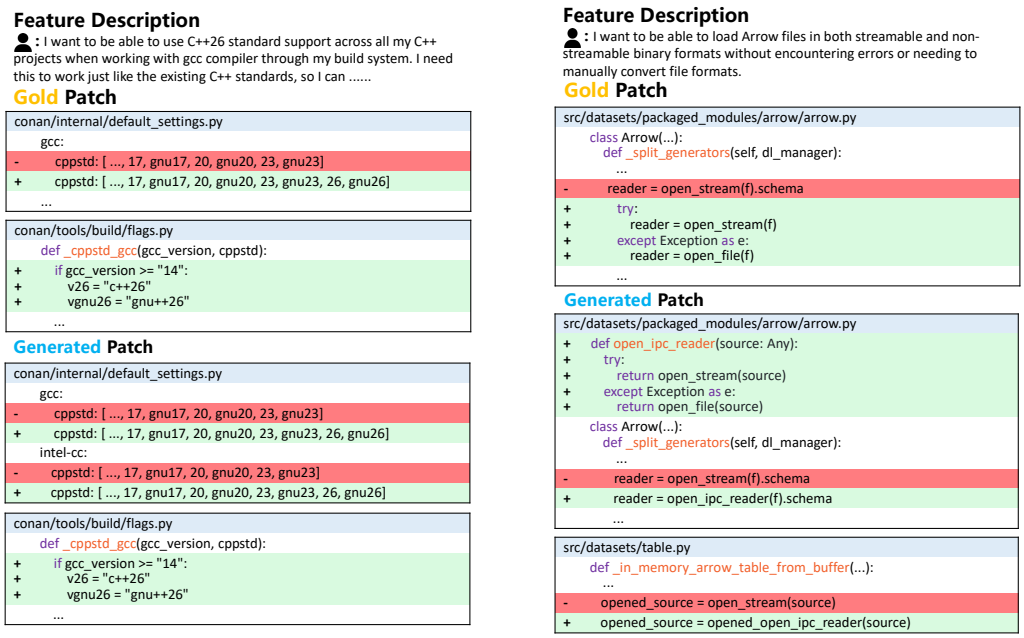


Figure 7: Association between Token Consumption and Resolved Rate

3.6 Token Consumption vs. Resolved Rate

Analysis of Fig. 7 reveals a stark trade-off between computational cost and task success.



(a) Wrong Generated Patch from conan-io/conan(Conan.io, 2025)

(b) Robust Gnerated Patch from huggingface/datasets(Lhoest et al., 2021)

Figure 8: Case Studies of Gold Patch and Generated Patch

The **Trae-agent** framework exhibits a near-linear trend where achieving higher resolved rates (15.92% to 29.94%) demands a proportionally significant token investment (1.07M to 2.90M). In contrast, the **Agentless** framework is highly token-efficient, consistently operating under 0.1M tokens, but this limits its success to a lower range of 7.00% to 16.56%.

This dichotomy highlights the critical need to find methods that enhance the **efficiency of token utilization**, ultimately increasing the resolved rate for complex agentic frameworks while avoiding a linear growth in token costs.

4 Case Studies

In this section, we present case studies of patches generated by Trae-agent.

4.1 Case Study on Failure Reasons

We manually analyzed 122 failed PRs from Trae-agent to identify the root causes of failure. These causes were categorized into three distinct types: (1) **Misunderstood User Intent**, where the agent fails to comprehend the core request; (2) **Incomplete Implementation**, where the agent understands the request but fails to meet all specific requirements; and (3) **Regressive Implementation**, where the agent successfully fulfills the request but inadvertently breaks existing functionality.

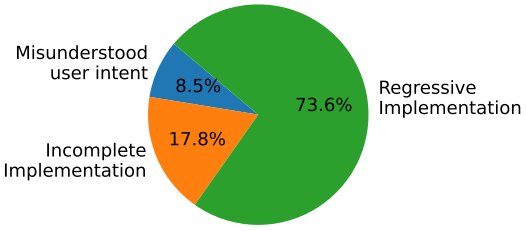


Figure 9: Failure Reasons of instances

As shown in Fig. 9, most failures fall into the third category. This indicates that while the agent is often capable of understanding and implementing the user's intent, its primary challenge lies in doing so without causing regressions that lead to test failures.

A clear example of **Regressive Implementation** can be seen in Fig. 8a. The objective was to add C++26 standard support for the GCC compiler. The gold patch correctly achieved this by updat-

ing both `default_settings.py` (to define the setting) and `flags.py` (to implement it). In contrast, the agent-generated patch adopted an **aggressive implementation**, attempting to also add C++26 parameters for the Intel C++ Compiler in `default_settings.py`. However, it neglected to add the corresponding implementation logic in `flags.py`. This created a configuration that was declared but not implemented, introducing a regression that caused previously passing tests reliant on `intel-cc` to fail.

This case illustrates the agent's tendency to proactively extend functionality beyond the user's explicit request, which can ultimately break the entire solution. This highlights a critical research direction: developing methods to ensure agents precisely address the specified problem without introducing extraneous, and potentially harmful, modifications. Preventing such "scope creep" is a key challenge for advancing agent-based coding.

4.2 Case Study on Resolved Patches: Agent vs. Gold

Our analysis also reveals that agent-generated resolved patches can outperform human-authored ones in robustness and generality, surpassing mere correctness. This is illustrated in Fig. 8b, which details a feature-request PR from the `huggingface/datasets` repository to support both streamable and non-streamable Apache Arrow file loading.

The human-written gold patch, while functional, employs a localized `try-except` block in `_split_generators`. This non-scalable, ad-hoc fix would require code duplication to support future streaming needs. In contrast, the agent-generated patch implements a more sophisticated architectural design. It abstracts the core logic into a reusable helper function, `open_ipc_reader(source: Any)`, encapsulating the stream-opening logic and its fallbacks. It then replaces direct calls with invocations of this centralized function in both `_split_generators` and `_in_memory_arrow_table_from_buffer`.

This design not only satisfies the immediate requirement but also establishes a scalable and maintainable pattern. It improves code modularity, prevents duplication, and simplifies future extensions of streaming support. This case demonstrates the agent's capacity to **move beyond basic problem-solving and produce qualitatively superior code from a software design perspective**.

4.3 The Duality of Aggressive Implementation

The two cases above indicate that the agent's tendency to adopt **aggressive implementation** strategies constitutes a **double-edged sword**. On one hand, as demonstrated by the C++26 support case (Fig. 8a), an overly aggressive approach can be detrimental, extending far beyond the original scope and introducing errors. On the other hand, this same impulse can produce superior software engineering outcomes. The Arrow file streaming case (Fig. 8b) illustrates how the agent's bold implementation led to thoughtful abstraction and refactoring.

Therefore, there is a critical need for mechanisms that can **control the agent's level of implementation aggressiveness**, allowing this trait to be harnessed for robust architectural improvements while preventing the harmful scope creep that introduces defects.

5 Conclusion

We introduce FeatBench, a novel benchmark designed to evaluate LLM-based coding agents on feature implementation within the vibe coding paradigm. Complementing existing benchmarks that primarily focus on bug fixing, FeatBench specifically targets feature implementation within the vibe coding paradigm. It utilizes code-free, natural language prompts to simulate authentic user-agent interactions. To combat data contamination, FeatBench developed a fully automated, evolving pipeline that will be dynamically updated. This paper releases its initial version, comprising 157 tasks from 27 diverse, real-world open-source repositories. Our experimental results reveal that feature implementation within the vibe coding paradigm presents a substantial challenge, with the highest resolved rates remaining below 29.94%. We uncover a tendency towards "aggressive implementation", which can lead to superior software design and critical failures. These insights help clarify the current coding agent limitations and can guide the development of more reliable agents for the emerging vibe coding paradigm.

Reproducibility Statement

All code, data, and experimental configurations associated with this research are publicly available at https://anonymous.4open.science/r/FeatBench-D3C5 to ensure full reproducibility. The repository includes detailed instructions and code to replicate our benchmark construction pipeline, execute the evaluation of the agents, and reproduce the results presented in this paper.

The complete methodology for our automated benchmark construction is detailed in Section 2.3 and further elaborated in the Section A.2, covering data curation, environment configuration, and test case validation. The experimental setup, including the specific agent frameworks, large language models, hyperparameters, and computational resources used for our experiments, is described in Section A.3. The prompts utilized for LLM interactions within our pipeline are provided in Section A.8. Researchers can fully reproduce our dataset and experimental findings by following the provided code and documentation.

Ethics Statement

Our work's primary goal is to advance the understanding and capabilities of LLM-based coding agents for beneficial vibe coding tasks, ultimately aiming to improve developer productivity.

Data Curation and Privacy. All data used to construct FeatBench was sourced exclusively from publicly available, open-source repositories on GitHub. No private or sensitive user data was collected.

Potential for Misuse. While our benchmark is designed to foster positive advancements in vibe coding scenarios, we recognize that the insights gained could potentially be applied for malicious purposes. However, the tasks in FeatBench are focused on vibe coding feature implementation, and our research does not inherently facilitate the development of harmful applications.

References

Anthropic. Claude 3.7 sonnet. https://www.anthropic.com/news/claude-3-7-sonnet, 2025.

Astral. Uv, 2025. URL https://docs.astral.sh/uv/.

ByteDance. Doubao-seed-1.6. https://console.volcengine.com/ark/region:ark+cn-beijing/model/detail?Id=doubao-seed-1-6, 2025.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

Conan.io. Conan, September 2025. URL https://github.com/conan-io/conan. original-date: 2015-12-01T13:17:02Z.

Deepseek. Deepseek v3.1. https://www.deepseek.com/, 2025.

Le Deng, Zhonghao Jiang, Jialun Cao, Michael Pradel, and Zhongxin Liu. Nocode-bench: A benchmark for evaluating natural language-driven feature addition. *arXiv preprint arXiv:2507.18130*, 2025.

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

Marko Horvat. What is vibe coding and when should you use it (or not)? *Authorea Preprints*, 2025.

Siyuan Jiang, Jia Li, He Zong, Huanyu Liu, Hao Zhu, Shukai Hu, Erlu Li, Jiazheng Ding, Yu Han, Wei Ning, et al. aixcoder-7b: A lightweight and effective large language model for code completion. *arXiv preprint arXiv:2410.13187*, 2024.

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

Andrej Karpathy. Andrej karpathy on x: "there's a new kind of coding i call "vibe coding", where you fully give in to the vibes, embrace exponentials, and forget that the code even exists. it's possible because the llms (e.g. cursor composer w sonnet) are getting too good. also i just talk to composer with superwhisper", 2025a. URL https://x.com/karpathy/status/1886192184808149383. Accessed: 25 September 2025.

Andrej Karpathy. Andrej karpathy on x: "the hottest new programming language is english", 2025b. URL https://x.com/karpathy/status/1617979122625712128. Accessed: 25 September 2025.

Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pp. 492–501, 2006.

Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Abhishek Thakur, Patrick von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Lewis Tunstall, Joe Davison, Mario Šaško, Gunjan Chhablani, Bhavitvya Malik, Simon Brandeis, Teven Le Scao, Victor Sanh, Canwen Xu, Nicolas Patry, Angelina McMillan-Major, Philipp Schmid, Sylvain Gugger, Clément Delangue, Théo Matussière, Lysandre Debut, Stas Bekman, Pierric Cistac, Thibault Goehringer, Victor Mustar, François Lagunas, Alexander Rush, and Thomas Wolf. Datasets: A community library for natural language processing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 175–184, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. URL https://aclanthology.org/2021.emnlp-demo.21.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Trans. Softw. Eng. Methodol.*, August 2024. ISSN 1049-331X. doi: 10.1145/3690635. URL https://doi.org/10.1145/3690635. Just Accepted.

OpenAI. Gpt-5. https://platform.openai.com/docs/models/gpt-5, 2025.

Pytest. Pytest. https://docs.pytest.org/en/8.0.x/, 2024.

Pytest-dev. Pytest-timeout, September 2025. URL https://github.com/pytest-dev/pytest-timeout.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, Tao Sun, Jinhua Zhu, Shulin Xin, Dong Huang, Yetao Bai, Lixin Dong, Chao Li, Jianchong Chen, Hanzhi Zhou, Yifan Huang, Guanghan Ning, Xierui Song, Jiaze Chen, Siyao Liu, Kai Shen, Liang Xiang, and Yonghui Wu. Seed-coder: Let the code model curate data for itself, 2025. URL https://arxiv.org/abs/2506.03524.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

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

Wikipedia. Vibe coding, September 2025. URL https://en.wikipedia.org/w/index.php?title=Vibe_coding&oldid=1312215031. Page Version ID: 1312215031.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint*, 2024.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Swe-bench goes live! *arXiv preprint arXiv:2505.23419*, 2025.

Yicong Zhao, Shisong Chen, Jiacheng Zhang, and Zhixu Li. Recode: Improving llm-based code repair with fine-grained retrieval-augmented generation. *arXiv preprint arXiv:2509.02330*, 2025.

A Appendix

A.1 The Use of Large Language Models

In this study, Large Language Models were solely employed for language refinement purposes to enhance the clarity, fluency, and academic rigor of the textual content presented. It is important to emphasize that all experimental design, data collection, result generation, and validation processes were conducted manually by human researchers to ensure the authenticity, reliability, and controllability of the experimental outcomes.

A.2 Construction Details

This section supplements Section 2.3 by providing technical details of our three-phase automated collection pipeline.

Data Curation and Pre-processing The data curation process begins with a four-stage filtering criteria applied at both the repository and release levels:

- Repositories must have at least three formal releases to ensure a development history.
- Repositories must have an identifiable test suite (e.g., `tests/` directory) to enable automated validation.
- Content relevance is ensured by excluding repositories focused on tutorials, examples, or similar non-production code.
- Timeliness is enforced by considering only releases published before June 1, 2024.

We construct a comprehensive prompt for each selected release that integrates the release title, description, and the repository's `README.md` file. We then instruct an LLM to categorize the contents of the release notes into `new_features`, `improvements`, `bug_fixes`, and `others`. While categorizing the content, we also ask the LLM to extract all relevant PR numbers from the text and associate them with the corresponding new feature entries.

After extraction, we apply a series of rule-based filters. We retain only those PRs that meet the following criteria:

- The PR must modify at least one Python file.
- The PR must include new or modified test cases to ensure the feature is verifiable.
- The *feature_patch* must only modify existing functions, without adding or deleting any function definitions.

Developers often write brief and ambiguous descriptions for their PRs in a real-world setting. To address this, we use an LLM to enrich the functional description of each PR. The model is prompted to analyze the original PR title, description, and associated code changes (*feature_patch*) to generate a comprehensive and clear natural language input. This augmented description serves as the final user request in our benchmark, simulating a vibe coding scenario.

Table 3: The required fields for our task instance. Fields marked with * are **newly added** in FeatBench compared to SWE-bench.

Field	Type	Description
base_commit	str	The commit on which the pull request is based, representing the repository state before the issue is resolved.
patch	str	Gold patch proposed by the pull request, in `.diff` format.
test_patch	str	Modifications to the test suite proposed by the pull request that are typically used to check whether the issue has been resolved.
problem_statement	str	Issue description text, typically describing the bug or requested feature, used as the task problem statement.
FAIL_TO_PASS	List[str]	Test cases that are expected to successfully transition from failing to passing are used to evaluate the correctness of the patch.
PASS_TO_PASS	List[str]	Test cases that are already passing prior to applying the gold patch. A correct patch shouldn't introduce regression failures in these tests.
*image_key	str	Instance-level docker image that provides an execution environment.

Environment Configuration Inspired by SWE-bench-Live, we construct a two-stage pipeline to configure the environment, mimicking the process of a human developer setting up an unfamiliar project. In the first stage, the analysis agent is tasked with information gathering. For each PR, it reverts the codebase to its corresponding *base_commit* state and, guided by a pre-configured prompt, locates crucial files such as CI/CD configuration files (e.g., `.github/workflows`, `.travis.yml`), dependency definitions (e.g., `requirements.txt`, `pyproject.toml`), and `README.md` files. It ultimately outputs a structured file containing the paths to the top 20 most relevant environment configuration files for the subsequent configuration agent to use.

The configuration agent operates within a container launched from a base image corresponding to the identified Python version. This base image is pre-equipped with common development tools (e.g., `git`, `cmake`). To accelerate dependency installation and reduce build times, the agent utilizes both a caching mechanism and the high-performance package manager `uv`(Astral, 2025). It is instructed to use the `--exclude-newer <commit_timestamp>` parameter to enforce the use of package versions available at the time of the PR's creation. The prompt provided to the agent also includes troubleshooting heuristics, such as installing `pytest-timeout`(Pytest-dev, 2025) to prevent tests from hanging and ignoring specific internal test framework errors. Following this process, the agent systematically completes the environment configuration, generating a "test-ready image".

Test Case Acquisition and Validation. In a container initiated from the "test-ready image," we first apply only the *test_patch*. We use Abstract Syntax Tree (AST) analysis on the *test_patch* to identify the specific tests added or modified for the new feature. We then selectively run these identified test cases by `pytest`(Pytest, 2024) and log the results, expecting them to fail. Next, we apply the *feature_patch* and rerun the same test cases, expecting them to pass. A PR is deemed valid if any test case transitions from FAILED/ERROR to PASSED.

In the container, we first apply the *test_patch* and run all tests within the main test directory, logging all passing cases. Subsequently, we apply the *feature_patch* and rerun all tests. These cases are used to check for regressions.

A.3 Experimental Setup Details

This section presents additional details of the experimental setup to facilitate reproducibility.

Hyperparameters used in the experiments. For Trea-agent, we set a maximum of 150 iterations per instance, with the LLM configured to use a temperature of 0.0 and a top-p value of 1.0 as the default. For Agentless, both the number of localization samples and repair samples are set to 1, corresponding to a single rollout. In our experiments, we omit Agentless's regression test-based reranking stage, retaining only the localization and repair stages. The LLM calls within FeatBench are configured with a temperature of 0.0.

Computational resources. All LLM calls in this work are made through official APIs. The experiments involve Docker containers for test execution. We conduct all the experiments on a desktop with an Intel Core i5-12600KF @ 3.70GHz (10 cores) and 16GB of RAM.

A.4 Task Instance Statistics

This section provides additional statistics about the instances in our benchmark.

Level	#Item	Average	Median
Repo	Repositories	27	
	LoC	209k	106k
	Files	864	381
Instance	Instances	157	
	Files*	2.4	2.0
	Hunks*	5.1	4.0
	Lines*	161.6	18.0
	F2P test cases	2.2	1.0
	P2P test cases	1692.4	1089.0

** Stats of gold patch.*

A.5 Benchmark Fields

Table 3 provides a detailed description of the fields included in the FeatBench and how they are obtained during the curation process.

A.6 Limitations

A limitation of our initial release is its exclusive focus on repositories predominantly written in Python. We prioritized Python because its widespread adoption ensures that our benchmark is representative of real-world vibe coding scenarios. Additionally, it is essential to note that our fully automated pipeline for data curation and environment configuration is language-agnostic. We plan to leverage this capability in future updates to incrementally incorporate repositories in other popular languages (e.g., Java, Go), thereby broadening the benchmark's scope and applicability across diverse programming domains.

A.7 FULL REPOSITORIES LIST

Type	Repository	License	#Instances	#Files	LoC
AI/ML	instructlab/instructlab	Apache-2.0	2	142	28.2k
	jupyterlab/jupyter-ai	BSD-3-Clause	1	81	9.0k
	stanfordnlp/dspy	MIT	7	222	30.6k
	projectmesa/mesa	Apache-2.0	5	109	20.3k
	huggingface/smolagents	Apache-2.0	8	65	21.4k
	cyclotruc/gitingest	MIT	1	39	4.7k
	modelcontextprotocol/python-sdk	MIT	2	114	13.4k
	openai/openai-agents-python	MIT	8	212	29.8k
	huggingface/datasets	Apache-2.0	6	207	69.6k
DevOps	conan-io/conan	MIT	56	1056	162.5k
	python-attrs/attrs	MIT	2	52	18.6k
	koxudaxi/datamodel-code-generator	MIT	8	599	60.3k
	tox-dev/tox	MIT	3	225	23.8k
	dynaconf/dynaconf	MIT	2	463	55.1k
	FreeOpcUa/opcua-asyncio	LGPL-3.0	1	168	344.4k
	iterative/dvc	Apache-2.0	4	554	85.3k
Web	reflex-dev/reflex	Apache-2.0	4	376	89.8k
	Kozea/WeasyPrint	BSD-3-Clause	1	144	70.0k
	aiogram/aiogram	MIT	6	861	69.8k
	ag2ai/faststream	Apache-2.0	3	1267	85.1k
	encode/starlette	BSD-3-Clause	6	66	17.2k
	jpadilla/pyjwt	MIT	3	26	6.9k
	slackapi/bolt-python	MIT	3	562	60.8k
Database	pydata/xarray	Apache-2.0	9	226	179.2k
Science	pybamm-team/PyBaMM	BSD-3-Clause	4	581	113.4k
Misc	fonttools/fonttools	MIT	2	512	192.6k
Cloud	aws-cloudformation/cfn-lint	MIT	2	2422	160.2k

A.8 PROMPTS IN FEATBENCH

Prompt for Feature-implementation PRs Identification

```
Repository Context (README):
{readme}

---

Analyze the following software release notes and categorize the
changes into: new_features, improvements, bug_fixes, and
other_changes.
For each change, extract any PR references (like #123, PR456, pull
#789, etc.) mentioned in the text.

Release version: {tag_name}
Release notes:{release_body}

Guidelines:
1. new_features: Brand new functionality, commands, rules, or
capabilities
2. improvements: Enhancements to existing features, optimizations,
performance improvements
3. bug_fixes: Bug fixes, error handling, crash fixes
4. other_changes: Documentation updates, dependency updates,
refactoring (only if significant)
5. Extract PR numbers from various formats: #123, PR #456, pull
789, (#101), etc.
6. Only include PR numbers that are explicitly mentioned with the
change
7. Ignore trivial changes like version bumps unless they're part
of larger features
8. Use the repository context to better understand the project's
domain and categorize changes more accurately

Return the result in JSON format:
{
    "new_features": [
        {
            "description": "Brief description of the new feature",
            "pr_ids": ["123", "456"]
        }
    ],
    "improvements": [
        {
            "description": "Brief description of the improvement",
            "pr_ids": ["789"]
        }
    ],
    "bug_fixes": [
        {
            "description": "Brief description of the bug fix",
            "pr_ids": ["101"]
        }
    ],
    "other_changes": [
        {
            "description": "Brief description of other changes",
            "pr_ids": []
        }
    ]
}
```

Prompt for Uesr Request Synthesization

```
You are creating a user requirement description that will be used
to instruct another LLM to implement the exact same functionality.
Your task is to analyze the PR information and code changes, then
write a comprehensive user request that describes what the user
wants to accomplish.

This description will be given to a coding LLM to generate the
implementation, so you must ensure all functionality across all
modified files is thoroughly described from the user's perspective.

Original Feature Description: {feature_description}

PR Title: {title}
PR Description: {body}

File Changes:
{files_text}

Your user requirement description must:
- Start with "I want to" and write as if a user is requesting this
functionality from a developer
- Include ALL information from the original PR Description - do
not omit any details, requirements, or context provided there
- Describe what the user wants to accomplish with complete detail
for every file that was modified
- Include all functional requirements that would be needed to
recreate this exact implementation
- When functions/methods have parameter changes (additions,
deletions, modifications), describe what the user needs in terms
of input data and configuration options, mentioning specific
parameter names naturally within the context of the requirement
- Focus on the complete user workflow and all capabilities they
need
- Describe the expected behavior and outcomes the user wants to
achieve
- Ensure a coding LLM reading this could implement all the
functionality without seeing the original code
- Write as a natural user request, not technical documentation
- Avoid phrases like "implement function X" or "modify file Y" -
instead describe what the user wants to accomplish
- Do NOT include any actual code implementations, code snippets,
or technical syntax - only describe the desired functionality and
behavior from a user perspective
- Focus on WHAT the user wants to achieve, not HOW it should be
implemented technically

Remember: This description will be the only guide for another LLM
to recreate this functionality, so include every important detail
about what the user wants to achieve, but express it as natural
user requirements. You must incorporate all information from the
original PR description. Never include code - only describe the
desired outcomes and behaviors.
```

Prompt for Relevant Files Identification & Python Version Detection

```
Please analyze the project {repo_name} and perform two tasks:

TASK 1: List the most relevant files(no more than 20) for setting
up a development environment, including:
0. CI/CD configuration files (such as .github/workflows/*.yml,
.gitlab-ci.yml, .travis.yml, Jenkinsfile, etc.)
1. README files
2. Documentation files
3. Installation guides
4. Development setup guides
5. Requirements and dependency files (requirements.txt, setup.py,
pyproject.toml, Pipfile, etc.)
6. Configuration files (.python-version, .nvmrc, Dockerfile,
docker-compose.yml, etc.)

Save the file list to a JSON file named "{setup_files}" in the
project root directory.
Format the file list as a JSON array where each element is a
string containing the relative path (relative to project root).
Example format: ["README.md", "requirements.txt", "setup.py",
".github/workflows/ci.yml"]

IMPORTANT: You MUST strictly save the file list using the exact
filename "{setup_files}" as required. Do NOT use any other
filename.

TASK 2: Analyze the configuration files you found and determine
the most appropriate Python version for this project.
Based on your analysis, save the recommended Python version to a
text file named "{version_file}" in the project root directory.
The file should contain ONLY the version number in the format
"3.xx" (e.g., "3.8", "3.9", "3.10", "3.11").

IMPORTANT: You MUST strictly save the Python version using the
exact filename "{version_file}" as required. Do NOT use any other
filename.

If no specific version requirements are found, use
"{default_version}" as the default.

IMPORTANT: The {version_file} file must contain ONLY the version
number, no comments, no explanations, no additional text.

Focus only on identifying files and determining the Python
version. Do not attempt to configure the environment yet.
```

Prompt for Configuration Agent

```
Please help me configure the runtime environment for this project.
The project is {repo_name}.

MANDATORY FIRST STEP
First, read the **{setup_files}** file in the project root to
understand which configuration files are available.
DO NOT PROCEED WITHOUT READING THIS FILE FIRST!
Then analyze these files to understand the project structure,
dependencies, and requirements.

TEST REQUIREMENTS
You need to verify if it can successfully run the tests of the
project.
You can tolerate a few test cases failures-as long as most tests
pass, it's good enough.
Your test command must output detailed pass/fail status for each
test item. This is mandatory. For example, with pytest, use the
-rA option to get output like:
PASSED tests/test_resources.py::test_fetch_centromeres
PASSED tests/test_vis.py::test_to_ucsc_colorstring

**IMPORTANT:** Always use '--tb=short' to avoid excessive
traceback output. Before running any tests, you MUST attempt to
install 'pytest-timeout' and 'pytest-xdist', and ALWAYS add
'--timeout=5' and '-n auto' to your pytest command to prevent any
single test case from running too long and to speed up testing by
running tests in parallel.

**IMPORTANT:** You must additionally test {test_files}. If these
specific tests fail due to ImportError or ModuleNotFound, you
should make every effort to install all optional dependencies. For
other test files, you may tolerate ImportError or ModuleNotFound
errors.

PYTHON VERSION INFORMATION
**IMPORTANT: About Python versions in this container:**
- The container has been pre-configured with the optimal Python
version for this specific project
- **IMPORTANT** The Python version specified in the {version_file}
is already installed in the system. You can use 'python3.x' to run
the project directly (e.g., python3.11)
- DO NOT attempt to install or change Python versions - the
correct version is already installed in the system

PREFERRED INSTALLATION METHOD
- Always check which Python environment you're targeting before
installation

**PRIORITY 1: Try using UV or poetry first (recommended)**
- Use 'uv' for dependency management and installation when possible
- For projects with pyproject.toml: try 'uv sync' or 'uv install'
- For projects with requirements.txt: try 'uv pip install -r
requirements.txt'
- For editable installation: try 'uv pip install -e .'
- For running tests: try 'uv run pytest' or similar commands
- **IMPORTANT**: If you use uv to install dependencies, you MUST
add the --exclude-newer {created_time} argument, for example: 'uv
pip install -r requirements.txt --exclude-newer {created_time}'
- **IMPORTANT**: Ensure uv is using the system Python, not the
agent's Python environment
```

```
**PRIORITY 2: Fallback to traditional tools if UV fails**
- If UV doesn't work or encounters issues, fall back to pip3
- Use system pip3: check with 'which pip3' to verify location
- Install necessary dependencies using system pip3
- If the project needs to be installed as a package, use 'pip3
install -e .'
- If you use pip to install dependencies, since pip cannot specify
a cutoff date, you may encounter dependency version conflicts. In
such cases, you need to manually resolve the conflicts according
to the project's timeline. The cut-off time is {created_time}.
- **IMPORTANT**: Verify you're using system pip3, not agent's pip

Set up the correct Python environment and ensure all required
packages are installed so that the test files can run properly.
Focus on resolving any import errors, missing dependencies, or
configuration issues.

IMPORTANT:
1. This Docker container may have been used to configure
environments for the same project before.
2. UV is already installed in the container and configured with
Tsinghua mirror for faster downloads.
3. If you encounter any testing-related errors (such as 'collected
0 items', 'no tests found', 'skip', 'INTERNALERROR',
'TypeError: could not get code object', or other pytest framework
internal issues), please ignore these errors and focus
on environment configuration instead. These errors may indicate
that test files don't exist, don't contain valid tests,
or are pytest framework internal issues. In such cases, abandon
trying to fix specific test files and continue with
dependency installation and environment configuration.
4. Install all dependencies to the system Python, not in any
virtual environment.
5. This Docker container may have been used to configure
environments for the same project before.
Please first check what packages are already installed in the
system Python environment to avoid conflicts.
```