GitGoodBench: A Novel Benchmark For Evaluating Agentic Performance On Git

Tobias Lindenbauer[1,2] [*], **Egor Bogomolov**[1], **Yaroslav Zharov**[1]
[1]JetBrains Research
[2]School of Computation, Information and Technology, Technical University of Munich
Correspondence: tobias.lindenbauer@jetbrains.com

Abstract

Benchmarks for Software Engineering (SE) AI agents, most notably SWE-bench, have catalyzed progress in programming capabilities of AI agents. However, they overlook critical developer workflows such as Version Control System (VCS) operations. To address this issue, we present GitGoodBench[1], a novel benchmark for evaluating AI agent performance on Version Control System (VCS) tasks. GitGoodBench covers three core Git scenarios extracted from permissive open-source Python, Java, and Kotlin repositories. Our benchmark provides three datasets: a comprehensive evaluation suite (900 samples), a rapid prototyping version (120 samples), and a training corpus (17,469 samples). We establish baseline performance on the prototyping version of our benchmark using GPT-4o equipped with custom tools, achieving a 21.11% solve rate overall. We expect GitGoodBench to serve as a crucial stepping stone toward truly comprehensive SE agents that go beyond mere programming.

1 Introduction

While the rapid scaling of Large Language Models (LLMs) has led to promising results across various tasks initially, the improvements gained from scaling models further are slowing down. Compared to GPT-3 (Brown et al., 2020), GPT-3.5 achieves a approximately 60% improvement (OpenAI et al., 2024a) on MMLU (Hendrycks et al., 2021). The improvement from GPT-3.5 to GPT-4, however, is just approximately 23% (OpenAI et al., 2024a). Scaling test-time compute rather than just models has emerged as an alternative for further improving performance, leading to the rise of AI agents (Yao et al., 2023; Shinn et al., 2023; Wang et al., 2024). AI agents equip LLMs with external tools (Schick et al., 2023) and employ sophisticated planning and reasoning strategies such as ReAct (Yao et al., 2023) or Reflexion (Shinn et al., 2023) to dynamically adjust in uncertain environments.

Software Engineering (SE) emerged as a pivotal application domain due to the availability of high-quality data in open-source repositories and because the creation and maintenance of software underpins innovation and economic impact across virtually every sector. SWE-bench (Jimenez et al., 2024) is the industry-standard benchmark for evaluating the agent's programming proficiency through testing the agent's ability to fix bugs in real-world software. This spurred the rapid development of AI agents for programming by major players in the tech tooling ecosystem (Cursor, 2024; Basu et al., 2024; Zakonov, 2025; Microsoft, 2025; Anthropic, 2025).

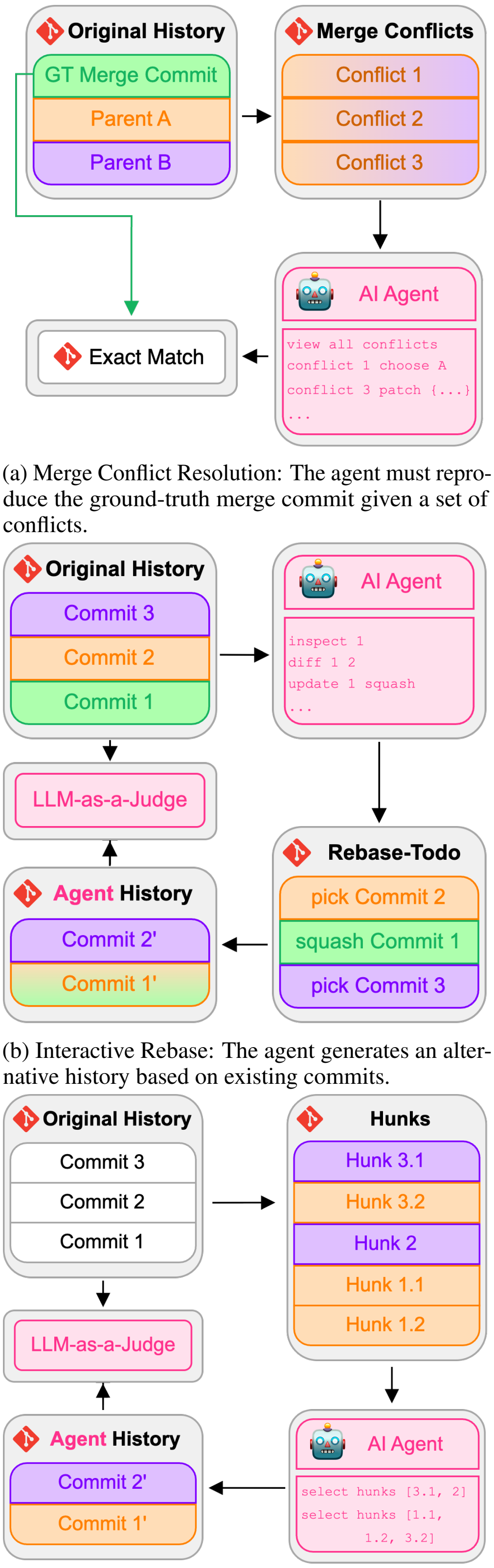
Version Control Systems (VCSs), such as Git, are ubiquitous in SE (Cortés Ríos et al., 2022) and play a pivotal role in building software in distributed teams. It is thus natural to use Git as a medium of collaboration between AI agents and human engineers. While LLM providers are advertising the Git capabilities of their systems (Anthropic, 2025), there currently exists no benchmark for evaluating an AI agent's capacity of interacting with Git in an end-to-end manner. Furthermore, typical Git tasks such as Interactive Rebase (IR) are time-consuming and distinct from raw code-generation. IR requires reasoning over the Git history and an in-depth understanding of dependencies between the commits constituting the history.

To stimulate innovation in the direction of comprehensive, end-to-end SE AI agents that go beyond mere programming, we introduce a novel benchmark for the popular VCS Git. This comprises a training corpus for collecting agentic trajectories and two evaluation sets (lite and full). The benchmark supports Merge Conflict Resolu-

[*]Work done during an internship at JetBrains

[1]https://github.com/JetBrains-Research/git-good-bench

Proceedings of the 1st Workshop for Research on Agent Language Models (REALM 2025), pages 272–288
July 31, 2025 ©2025 Association for Computational Linguistics



```
view all conflicts
conflict 1 choose A
conflict 3 patch {...}
...
```

(a) Merge Conflict Resolution: The agent must reproduce the ground-truth merge commit given a set of conflicts.

```
inspect 1
diff 1 2
update 1 squash
...
```

(b) Interactive Rebase: The agent generates an alternative history based on existing commits.

```
select hunks [3.1, 2]
select hunks [1.1,
          1.2, 3.2]
```

(c) Iterative Committing of Changes: The agent generates an alternative based on a disorganized set of changes. We only use the original commit history for evaluation.

Figure 1: The three Git scenarios supported by GitGoodBench. Each scenario benchmarks a typical Git use-case and unique aspect of version control.

tion (MCR), Interactive Rebase (IR), and the Iterative Committing of Changes (ICC) (Figure 1). We scrape all data from permissive, open-source, Python, Java, or Kotlin GitHub repositories. Furthermore, we provide a baseline implementation using GPT-4o (OpenAI et al., 2024b) with custom tools, achieving a 21.11% solve rate.

2 Related Work

Several benchmarks, such as SWE-bench (Jimenez et al., 2024), or the Kowinski prize (Konwinski et al., 2024) evaluate agentic systems on complex, multi-turn SE tasks sourced from real-world GitHub issues. While the environment allows Git usage, the evaluation focuses solely on whether the agent resolves the bug rather than how it leverages VCS. In contrast, our benchmark explicitly measures an agent's proficiency with Git tasks. This allows future research to thoroughly examine and refine VCS-focused strategies in SE agents and tailor agents to VCS tasks specifically.

While previous works on automating or evaluating MCR (Svyatkovskiy et al., 2022; Shen et al., 2023; Boll et al., 2024; Pan et al., 2021) and commit message generation or completion (Jiang et al., 2017; Hal et al., 2019; Eliseeva et al., 2023) exist, they exclusively cater to specific VCS subtasks. In contrast, our benchmark is the first to encapsulate multiple subtasks, such as commit message generation, reasoning across commits, and rebase plan generation into a single benchmarking scenario. This uniquely positions GitGoodBench for evaluating and training AI agents with expertise in VCS tasks in end-to-end settings.

3 GitGoodBench Datasets

We provide GitGoodBench (900 samples) and GitGoodBench Lite (120 samples) for evaluation in comprehensive and rapid-prototyping settings, respectively. The research community recently started investigating SE agents powered by fine-tuned Small Language Models (SLMs) (Pan et al., 2024; Jain et al., 2025; Yang et al., 2025). We believe that trained, on-device sized agents are an exciting research direction. While we do not train such a model in this work, with GitGoodBench Train (17,469 samples) we release a dataset split dedicated to collecting trajectories for training Git agents.

Filter	Reason
Repository size $\leq$ 400MB	Avoid Git LFS heavy repositories
Repository stars $\geq$ 1000	Heuristic for repository quality
Repository is not archived	Heuristic for repository quality
Repository is not forked	Avoid duplication
Last commit within a month of May, 31st 2024	Avoid stale repositories
Repository has permissive license	Ensure legal compliance
Repository $\geq$ 5 branches	Heuristic for merge conflict scenarios
Repository $\geq$ 5 contributors	Heuristic for merge conflict scenarios

(a) Repository metadata filters we use for selecting the initial repositories we consider in the benchmark creation. We consider the following licenses permissive: MIT, Apache 2.0, BSD 3-Clause "New" or "Revised", BSD 2-Clause "Simplified".

Filter	Reason
No merge commit in File-Commit Chain (FCC)	Cleanly separate scenario types
No merge conflict in unsupported file type	Only support Python, Java, or Kotlin
All merge scenarios contain conflict	Merges without a conflict are trivial
Merge scenarios have exactly two parents	Avoid dilution by complex and rare merge types
Number of merge conflicts $\leq$ 8	Ensure the agent can theoretically solve the scenario
Length of FCC $\leq$ 6	Ensure the agent can theoretically solve the scenario
FCC file is modified, not added	Otherwise we get a single hunk when resetting

(b) Scenario level filters for selecting scenarios to include in our benchmark.

Table 1: Filters for selecting repositories and scenarios to include in our benchmark.

3.1 Supported Scenarios

Our benchmark covers the following three types of Git scenarios:

Merge Conflict Resolution The agent must resolve all merge conflicts by reproducing the ground truth resolutions (Figure 1a).

Interactive Rebase In this scenario (Figure 1b) the agent must reason across commits and their contents to determine the optimal ordering of commits, thereby improving the Git history. This includes commit consolidation or modification and commit message refinement.

Iterative Committing of Changes This scenario (Figure 1c) type is the inverse of the IR. Instead of optimizing existing commits, the agent must generate a reasonable Git history from a large disorganized set of changes.

With these scenario types we cover non-trivial Git functionalities central to common Git workflows (Cortés Ríos et al., 2022). Moreover, we explicitly cover functionality currently only implemented interactively in Git (e.g., `git rebase -i` or `git add -p`). Agents are highly applicable for such iterative tasks that depend on environment observations. However, interacting with such functionality is challenging for agentic systems because these functions do not provide immediate feedback and instead wait for user input. This introduces friction into the typical plan-act-observe loop of AI agents, due to delayed feedback not easily captured by usual pipelines.

3.2 Dataset Creation

We collect repository metadata from repositories with permissive licenses using SEART (Dabic et al., 2021) and the metadata filters defined in Table 1a. The scenarios for IR and ICC are represented by the same samples in our dataset (i.e., with one sample, we can evaluate both IR and ICC). We call these samples File-Commit Chain (FCC) samples and they refer to chains of commits in Git histories in which we observe consecutive modifications of a single file. We use this as a heuristic for identifying Git histories that may be improved through reordering or consolidating commits. These samples target the use-case of (1) cleaning up the local Git history before pushing new commits to the remote (e.g., `git rebase -i HEAD~5`, and (2) constructing a clean git history given a set of changes for the IR and ICC scenario, respectively. To tailor these samples toward evaluating an aspect of Git distinct from MCR, we remove merge commits from FCCs. This allows us to evaluate the system's understanding of the rebase-todo and of relationships between commits. We then mine the Git history of these repositories for merge, and FCC samples and apply our scenario-level filters (Table 1b) to obtain 6,917 merge samples and 11,572 FCC samples. To ensure a diverse benchmark, especially concerning represented repositories, we partition our data into strata based on the following features before sampling to construct our benchmark.

File-Commit Chain Samples For these samples, we use the project size (in lines of code) and the repository name for stratification.

Merge Conflict Resolution Samples In addition to the above, we stratify on the difficulty of these samples. We define MCR difficulty based on the number of conflicts and their distribution across files. To determine conflicts, we run `git show -remerge-diff <merge-commit>` and identify conflicts through Git merge conflict markers. We consider scenarios with a single conflict "easy" because no reasoning across diffs is necessary, those with multiple conflicts in a single file "medium" because reasoning across diffs in the context of a single file is required, and all others, for which the agent must reason across multiple diffs and files, as "hard".

To construct the held-out test, we sample 120 scenarios for GitGoodBench Lite and 900 for GitGoodBench. We stratify the sampling for scenario type and Programming Language (PL). The remaining samples yield GitGoodBench Train. All three datasets are mutually disjoint. For further details, see Appendix A.

3.3 Metrics

We present the results of our baseline in terms of success and solve rate (both expressed as percentages). The *success rate* refers to scenarios for which our system did not cause an error (e.g., because a patch cannot be applied in MCR). Below, we define the *solve rate* for each scenario:

File-Commit Chain Samples For FCC scenarios we prompt an LLM to judge the agent-generated and ground truth Git histories using the LLM-as-a-Judge (Zheng et al., 2023) approach. We opt for this approach instead of Exact-Match (EM), because there is no clear, deterministic way to define what constitutes a superior Git history. Following Zheng et al. (2023) we judge each pair of Git histories twice while switching the positions of the histories in the same prompt template to account for position bias. We prompt the judge to base its decision on (1) the quality of the commit messages considering the contents of the commit, (2) the cohesion of changes within the commits, (3) a logical progression of changes across commits, and (4) the size of commits. If the judge chooses the agent-generated over the ground truth Git history in both cases, we count a sample as solved. For details on the prompt see Appendix B.4.

Scenario	Success Rate	Solve Rate
IR	93.33	26.67
ICC	93.33	23.33
MCR	76.67	13.33
Total	88	21.11

Table 2: Success and solve rates (%) by scenario type, rounded to two decimal places. We observe the high complexity of the proposed benchmark, even given the strong baseline model and custom environment tools.

Difficulty Level	Success Rate	Solve Rate
Easy	80.64	22.58
Medium	84.62	7.69
Hard	62.5	0

Table 3: Success and solve rates (%) by difficulty for MCR samples, rounded to two decimal places. GitGoodBench Lite contains 31 ($\approx 52\%$) easy, 13 ($\approx 22\%$) medium, and 16 ($\approx 27\%$) hard samples.

Merge Conflict Resolution Samples Because an exact ground truth solution is available, we use EM between the ground truth solution and the agent's solution for evaluating MCR.

4 Environment

As a baseline, we evaluate GPT-4o (OpenAI et al., 2024b) on GitGoodBench Lite and the tasks defined in Section 4.1 using the metrics in Section 3.3. While we do not use an agentic reasoning framework (Yao et al., 2023; Shinn et al., 2023; Wang et al., 2024), we do equip the LLM with one possible set of custom tools (Section 4.2).

4.1 Provided Context

Interactive Rebase In the initial context, we provide all changes in all commits participating in the IR, few-shot function-calling examples and an explanation of valid commands for the rebase-todo file. We initiate the IR covering all commits in the FCC before launching the agent.

Iterative Committing of Changes We provide all Git-generated hunks that the agent must process, in addition to few-shot function-calling examples in the initial context. After each commit, we automatically show the agent the updated list of remaining hunks. We limit the agent's selection of hunks to hunks originating from the file for which we mined the FCC and commit all other changes in a single commit after the agent terminates.

Merge Conflict Resolution The initial context includes the temporal ordering of the commits being merged, names of all files with conflicts and all merge conflicts it must resolve as well as few-shot function-calling examples.

4.2 Provided Tools

Initially we experimented with minimalistic tooling, simply giving the LLM terminal access in a sandbox environment. However, preliminary results indicated that the system is unable to make any meaningful progress in this setup[2]. In particular it struggled with interactive Git functionality (Section 3.1. Because of this we opt for the strong scaffolding detailed below.

Interactive Rebase We implement tools for viewing the contents of commits and interacting with the rebase-todo list, a file that specifies how Git should carry out the IR.

Iterative Committing of Changes With our tooling for this scenario type, the agent selects any number of Git-generated hunks to group into a single commit.

Merge Conflict Resolution To foster coherent, conflict-spanning resolutions, we provide tools for viewing individual merge conflicts, complete files or the overall difference between commits being merged. Our tooling limits the agent to sequentially resolving conflicts. It may only specify a patch for resolving the current conflict.

5 Baseline Results

In Table 2, we see that our baseline implementation succeeds in 88% and solves 21.11% of scenarios in GitGoodBench Lite[3] overall. Even with significant scaffolding support the LLM is unable to solve the majority of tasks in our benchmark. This highlights the need to explicitly consider Git use-cases when engineering and training SE agents.

For both IR and ICC scenarios our system achieves higher success and solve rates than for MCR scenarios (Table 2). We partially attribute to the stricter scaffolding for these two scenarios. In MCR scenarios the agent must generate code that can be applied at the location of the conflict to solve the conflict. Especially in scenarios which require the agent to make globally consistent conflict resolution choices (i.e., medium and hard samples in Table 3) the system's performance rapidly deteriorates. In FCC-based scenarios, the agent must simply select a set of hunks to commit for ICC scenarios or modify the rebase-todo file through a tool for IR scenarios. This indicates that the failure rate of agentic systems interacting with Git increases as the level of technical abstraction from Git decreases. We do however note that some amount of this performance degradation may also be due to the stricter EM evaluation metric used for MCR scenarios. Regarding the difficulty heuristic for MCR, we note that it accurately captures a sample's complexity regarding the solve rate. Easy samples have a $\approx$ 3 times higher solve rate than hard samples. Furthermore, the scenarios based on FCC samples (IR and ICC) result in similar success and solve rates. This indicates that our LLM-as-a-Judge evaluation methodology is consistent in assessing similar Git histories and is thus a suitable choice. Our difficulty heuristic for IR and ICC scenarios did not correlate with the observed difficulty, for details see Appendix A.2.3.

6 Conclusions

GitGoodBench is a novel benchmark for training and evaluating AI agents on the Git scenarios: MCR, IR and ICC. Our baseline implementation demonstrates capabilities in resolving merge conflicts and improving Git histories when equipping GPT-4o (OpenAI et al., 2024b) with tools for interacting with Git, achieving an overall solve rate of 21.11% on GitGoodBench Lite. The poor overall performance and the observed performance degradation for MCR across difficulty levels highlight the need to explicitly consider Git when designing SE agents. Just as we construct agents for SE with repository-level reasoning and code generation in mind, we should consider the agents' understanding of Git artifacts and capacity to use Git functionality. We hope our benchmark spurs innovation in this direction.

7 Limitations

Our baseline implementation has several constraints that present opportunities for improvement. The MCR tooling cannot modify Git-generated hunk boundaries, limiting flexibility when these hunks are too coarse. For ICC, expanding be-

[2] We acknowledge that a Git Model-Context Protocol (MCP) may address this issue but as the focus of our work is a benchmark, we do not further investigate this.

[3] We release the raw evaluation data with our repository.

yond a single-file focus would allow more accurate handling of multi-file changes. Furthermore, enabling commit content modification during IR would allow handling more complex IR scenarios, including ones during which a merge conflict occurs. Additionally, for FCC samples our evaluation methodology may introduce bias, as it is LLM-based. We suggest that future work evaluating agents on GitGoodBench use an ensemble of LLMs for judging trajectories to mitigate bias and subjectivity of the evaluation. Finally, we did not investigate how a Git implementation of the novel Model-Context Protocol (MCP) (Anthropic, 2024) affects an agent's ability to solve Git tasks.

Regarding the dataset itself, while we made efforts to ensure diversity, certain limitations remain. While our difficulty heuristic for MCR showed promising results, a FCC difficulty heuristic based on FCC purity (Appendix A.2.3) didn't correlate with empiric performance. Due to this, the distribution of FCC samples may be skewed with respect to their difficulty in our benchmark. While our three scenario types cover core Git functionality, our benchmark does not yet include important Git diagnostic workflows such as `git bisect`. Incorporating bisect scenarios would enable evaluation of an AI agents' ability to systematically locate commits introducing bugs, a capability that could significantly enhance automated debugging and regression analysis in SE AI agents. Furthermore, as our benchmark is static, we may need to update our benchmark with more diverse and complex scenarios to counteract benchmark saturation and data leakage.

Acknowledgements

We thank Yury Khudyakov, Alexandra Eliseeva, Maria Tigina, and Abhiram Bellur for the valuable discussions and advice during this project.

References

Anthropic. 2024. Introducing the model context protocol. Accessed on May 20, 2025.

Anthropic. 2025. Claude 3.7 sonnet and claude code. Accessed on February 27, 2025.

Shrestha Basu, Mallick, and Kathy Korevec. 2024. The next chapter of the gemini era for developers. Accessed on February 27, 2025.

Alexander Boll, Yael Van Dok, Manuel Ohrndorf, Alexander Schultheiß, and Timo Kehrer. 2024. Towards Semi-Automated Merge Conflict Resolution: Is It Easier Than We Expected? In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, pages 282–292.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Julio César Cortés Ríos, Suzanne M. Embury, and Sukru Eraslan. 2022. A unifying framework for the systematic analysis of Git workflows. *Information and Software Technology*, 145(C).

Cursor. 2024. New composer ui, agent, commit messages. Accessed on February 27, 2025.

Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 560–564.

Aleksandra Eliseeva, Yaroslav Sokolov, Egor Bogomolov, Yaroslav Golubev, Danny Dig, and Timofey Bryksin. 2023. From Commit Message Generation to History-Aware Commit Message Completion. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

S. R. P. van Hal, M. Post, and K. Wendel. 2019. Generating Commit Messages from Git Diffs. *arXiv preprint*. ArXiv:1911.11690 [cs.SE].

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. In *International Conference on Learning Representations*.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2E-Gym: Procedural Environments and Hybrid Verifiers for Scaling Open-Weights SWE Agents. *arXiv preprint*.

Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 135–146.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R

Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Andy Konwinski, Christopher Rytting, Justin Fiedlerand Alex Shaw, Sohier Dane, Walter Reade, and Maggie Demkin. 2024. Konwinski prize. `https://kaggle.com/competitions/konwinski-prize`. Kaggle.

Microsoft. 2025. Introducing github copilot agent mode for vscode. Accessed on February 27, 2025.

OpenAI et al. 2024a. GPT-4 Technical Report. *arXiv preprint*. ArXiv:2303.08774 [cs.CL].

OpenAI et al. 2024b. Openai gpt-4o system card. Accessed on March 6, 2025.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training Software Engineering Agents and Verifiers with SWE-Gym. *arXiv preprint*. ArXiv:2412.21139 [cs].

Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu Lahiri, and Mike Kaufman. 2021. Can Program Synthesis be Used to Learn Merge Conflict Resolutions? An Empirical Analysis. In *Proceedings of the 43rd International Conference on Software Engineering*, pages 785–796.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Chaochao Shen, Wenhua Yang, Minxue Pan, and Yu Zhou. 2023. Git Merge Conflict Resolution Leveraging Strategy Classification and LLM. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pages 228–239.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pages 8634–8652.

Alexey Svyatkovskiy, Sarah Fakhoury, Negar Ghorbani, Todd Mytkowicz, Elizabeth Dinella, Christian Bird, Jinu Jang, Neel Sundaresan, and Shuvendu K. Lahiri. 2022. Program merge conflict resolution via neural transformers. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 822–833.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. In *Proceedings of the 41st International Conference on Machine Learning*, pages 50208–50232.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Andrew Zakonov. 2025. Meet junie, your coding agent by jetbrains. Accessed on February 27, 2025.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems*, volume 36, pages 46595–46623.

A Dataset Details

In this section we provide further details about the diversity of our datasets with respect to represented repositories, and MCR difficulty. For GitGoodBench Train we also provide information on the distribution across programming languages, for all other datasets this distribution is fixed to ensure diversity (see Section 3.2). Please also refer to our dataset cards on HuggingFace: GitGoodBench Lite, GitGoodBench, GitGoodBench Train.

In Table 5 we provide statistics on the diversity of our datasets with respect to the repositories represented. Notably, there is a heavy skew toward Python and to a lesser extent Java. However, this is in line with our expectations and the popularity of the programming languages we consider in our datasets. Table 6 provides further information regarding the distribution of MCR difficulties across our datasets. We note that the difficulty of MCR is overall relatively well-distributed with a spike in difficulty on GitGoodBench. Despite stratifying on difficulty, these spikes can occur because we also stratify on other features such as the programming language.

A.1 Sample Data

Table 7 shows the complete structure of a data point in our dataset. The detailed contents of the `scenario` field vary depending on the `sample_type` and are presented in Appendix A.2.

Scenario Type	Easy	Medium	Hard	Success Rate	Easy	Medium	Hard	Solve Rate
IR	100	86.36	95.52	93.33	13.33	31.82	30.43	26.67
ICC	100	90.91	91.3	93.33	20	27.27	21.74	23.33
MCR	80.64	84.62	62.5	76.76	22.58	7.69	0	13.33
Total	90.16	87.72	85.48	88	19.67	24.56	19.35	21.11

Table 4: Success and solve rates (%) by scenario type and difficulty, rounded to two decimal places. GitGoodBench Lite contains 31 easy, 13 medium, 16 hard MCR samples and 15 easy, 22 medium, and 23 hard FCC samples.

Statistic	GitGoodBench Lite	GitGoodBench	GitGoodBench Train
Total Repositories	100	479	816
Mean Samples Per Repo	1.20	1.87	21.40
Standard Deviation	0.79	2.80	48.80
Minimum	1	1	1
25th Percentile	1	1	2
Median	1	1	6
75th Percentile	1	2	18
Maximum	8	46	644

Table 5: The diversity of our datasets with respect to unique repositories from which we mined our samples. Our datasets consist of 816 (525 Python, 284 Java, and 79 Kotlin) unique repositories overall.

A.2 The Scenario Field

In this section we provide further details regarding the contents of the `scenario` for the two sample types in our datasets.

A.2.1 Contents For FCC Samples

In Table 8 we show the structure of the scenario field for FCC samples. Furthermore, Table 11 provides an exemplary FCC datapoint's `scenario` field contents. The `scenario` contains information regarding the source of the sample (e.g., the branch from which it was mined), the length of the FCC and its starting and end commits.

A.2.2 Contents For Merge Samples

In Table 8 we detail the structure of the scenario field for MCR samples. Table 10 shows a representative example of a MCR `scenario` field from our GitGoodBench Lite. The `scenario` field contains the metadata based on which we compute the difficulty of this sample. In this case, the sample is hard, because there are multiple conflicts across multiple files. Furthermore, the sample contains the merge commit that serves as ground truth. We use the parent commits of this merge commit to generate a merge conflict that is resolved in the merge commit.

A.2.3 File-Commit Chain (FCC) Difficulty Heuristic

For FCC scenarios we define their difficulty through the purity of the FCC:

$$d_{FCC}(p) = \begin{cases} \text{pure}, & 0.5 \leq p < 0.75 \\ \text{mixed}, & 0.75 \leq p < 1 \\ \text{noisy}, & \text{otherwise} \end{cases}$$

where p refers to the ratio of changes within the file for which we mined a FCC to the overall changes in a FCC. We consider any line with a Git change prefix (+ or -) a change.

We expected this to capture the difficulty of IR and ICC scenarios, because it captures how distributed changes the agent has to reason with are across files. The intuition being that it is easier to coherently generate commits and a plan for rebasing, when the reasoning spans fewer files. While, for FCC samples easy scenarios exhibit the maximum *success rate* of 100%, they have a lower *success rate* than medium and hard scenarios (Table 4). One possible explanation could be that we are simply considering the ratio of changes and not the overall number of changes. A large overall number of changes forces the agent to reason across a much larger context window than a smaller number, yet in the purity-based difficulty heuristic we investigated, both are assigned the same difficulty.

Difficulty	GitGoodBench Lite	GitGoodBench	GitGoodBench Train
Easy	51.67	41.33	51.65
Medium	21.67	24.44	18.39
Hard	26.67	34.22	29.97

Table 6: Difficulty distribution (in %) across GitGoodBench datasets.

Field	Value	Description
id	`mockito_mockito_merge_0002`	Unique identifier
name	`mockito/mockito`	Repository name (owner/repository)
default_branch	`main`	Primary repository branch
license	`MIT License`	Repository license
stargazers	14,617	GitHub stars count
created_at	`2012-10-13T08:27:12`	Repository creation date
topics	`java;java-library;mock;...`	Repository topics/tags
programming_language	`java`	Primary language
scenario	`<scenario-details>`	Scenario-specific data (see Tables 10 and 11)
sample_type	`merge`	Type of code sample
project_size	`medium`	Estimated project size
difficulty	`easy`	Complexity level

Table 7: Structure of a sample data point from our dataset. Each entry contains metadata about the repository, along with scenario-specific information that varies based on the sample type. The `topics` field is truncated for brevity.

Field	Description
`file`	The relative path of the file this sample refers to.
`branch`	The branch name from which this FCC originates.
`times_seen_consecutively`	The number of times this particular file was modified in succession.
`purity`	$\in [0; 1]$. Ratio between changes in the `file` and the total changes in all files of a FCC scenario.
`newest_commit`	The commit hash corresponding to the newest or last commit in this FCC.
`oldest_commit`	The commit hash corresponding to the oldest or first commit in this FCC.
`contains_non_pl_files`	A boolean indicating whether any commit in this sample includes changes to files with types not covered by the supported PLs.

Table 8: Contents For FCC Samples. Table 11 shows a representative example of the scenario field from our dataset. Due to a `purity` of 0.68, we consider this sample to be of medium difficulty. We define the purity-based difficulty we investigated in more detail in Appendix A.2.3.

B Prompts

In this section we will provide the prompts used by our system for the individual scenarios and the LLM-as-a-Judge evaluation. For any missing details please refer to our repository.

B.1 Merge Conflict Resolution (MCR) Scenarios

In Figures 2 to 4 we provide the prompt our system uses for MCR scenarios. We show information on (1) the temporal ordering of the merge parent commits, (2) which conflicts occur (`git show` output) and (3) detailed instructions for resolving conflicts. Furthermore, we provide examples for the tools we provide in various conflict resolution contexts.

Field	Description
`merge_commit_hash`	The ground truth merge commit in which the conflicts are resolved.
`parents`	List of parent commit hashes of the merge commit.
`number_of_files_with_merge_conflict`	The overall number of distinct files in which a merge conflict occurs.
`total_number_of_merge_conflicts`	Total number of distinct merge conflicts across all files.
`files_in_merge_conflict`	Relative paths of the files that contain merge conflicts.

Table 9: Contents For Merge Conflict Resolution (MCR) Samples.

Field	Value
merge_commit_hash	`baa37f65fdff5b780a50d5b5c6bf8bc3ade43815`

parents	`[d758810c59a9134f437d60f73a82036749688ccb,`
	`5dcd493c67ff863c69c1214f0892a80e4951087e]`

number_of_files_with_merge_conflict	2
total_number_of_merge_conflicts	2

files_in_merge_conflict	`[cogs/gpt_3_commands_and_converser.py,`
	`models/openai_model.py]`

Table 10: A sample Merge Conflict Resolution (MCR) scenario field from GitGoodBench Lite. Each entry contains metadata about a specific merge conflict instance, including commit identifiers and statistics about the conflicting files.

B.2 Interactive Rebase (IR) Scenarios

In Figures 5 to 7 we provide the prompt our system uses for IR scenarios. We provide information on the commits participating in the rebase (`git show` output) to save agent turns spent reading the commit information. Then we provide detailed instructions for performing an interactive rebase. Finally, we provide examples for the tools we provide and the JSON schema the agent must use to interact with the rebase-todo file.

B.3 Iterative Committing of Changes (ICC) Scenarios

In Figures 8 and 9 we provide the prompt our system uses for ICC scenarios. First, we provide detailed instructions for chunking changes into logically cohesive commits that incrementally build toward the final patch. Next, we show the contents of the hunks the agent can select to save agent turns spent reading the commit information. Finally, we provide examples for the tools that the agent can use in these scenarios.

B.4 LLM-as-a-Judge Evaluation

In Figures 10 and 11 we provide the prompt our system uses when evaluating the Git histories generated by the agent in FCC samples. First, we provide detailed instructions regarding the dimensions based on which the LLM should assess the quality of a history. Next, we show the model one example response for each evaluation case. By doing so, we help the model follow the response schema. We also specify the response schema directly in the model configuration. Finally, we present the ground truth and agent-generated Git history. We use the same prompt for both evaluation runs when re-evaluating to mitigate the position bias.

Field	Value
file	composer/models/huggingface.py
branch	origin/vincent-mlflow-logger-verbose
times_seen_consecutively	3
purity	0.68
newest_commit	c24b29f19c4c131a3ea7098dd8b8a5edde344819
oldest_commit	c1ff80900f46d4e36feb4b326689fe14fc41cbc6

Table 11: A sample File-Commit Chain (FCC) scenario field from GitGoodBench Lite. This example records a file's modification pattern across multiple commits, including branch information and a purity metric defined in Appendix A.2.3 and Section 3.1.

Merge Conflict Resolution (MCR) Prompt - Part 1

```
You are a staff software engineer with expertise in {programming_language} and git.

You are helping a junior team member who has initiated a merge that resulted in
one or more merge conflicts in one or more files. Your task is to help your
junior colleague with resolving all {total_amount_of_merge_conflicts} merge
conflicts.

The semantic meaning and temporal relationship of the two sides of the merge
conflicts are as follows for ALL merge conflicts you will encounter:
{commit_temporal_ordering}

The following files have merge conflicts:
{files_with_conflicts}

Below are all merge conflicts that need to be resolved, delimited by <CONFLICT-i>
tags where i is the 0-based index:
{all_merge_conflicts}
```

Figure 2: Our MCR prompt.

MCR Prompt - Part 2

```
Instructions:
- Start with resolving the conflict at index 0 (CONFLICT-0) and proceed in
ascending order through the conflicts.
    CONFLICT-0 is the current conflict that needs to be resolved.
- Consider the context around the merge conflicts, of the overall diffs and files
in which the conflicts occur.
- Resolve the conflicts in a cohesive manner. For example, if you remove a function
in a conflict, make sure that you also remove any invocations of that function in
any other conflicts.
- If you are just choosing one of the two sides, without changing any of the actual
content, make sure to also reproduce the whitespaces exactly.
- If the merge conflict occurs due to a NOP (e.g. one side of the conflict is empty,
the other is a commented code block) favor resolving the conflict to the most
maintainable and concise way. Avoid dead code.
- Make sure to consider the implications your previous resolutions have on the
remaining resolutions, especially when resolving multiple conflicts in a single
file.
- If you find simple bugs, such as typos, copy and paste errors in variable
assignments or parameters, feel free to help your junior developer fix these.
Do not perform complex refactorings or attempt to change code drastically.
Make as few changes to the side that you are accepting as possible.
- Consider the context of the temporal relationship of the branches that are being
merged and the intent of the junior developer, with respect to which side of the
conflict contains the local and which the incoming changes. The intent of the
developer is to merge the incoming changes into the local changes.

You must only use the following tools and follow their specification exactly and
always provide a reason for calling a tool.

All tools other than the ones listed below are INVALID and you MUST NOT use them
under any circumstances.

Valid tools:
- view_current_merge_conflict_with
- view_merge_conflict_at
- resolve_current_merge_conflict_with
- view_diff_for
- view_file_at: You must not use this command more than once per file as it is costly.

Below follow some examples detailing the usage of the above tools:
view_current_merge_conflict_with(context_window_size=15, reason='to get a more
comprehensive overview of the local context around the current merge conflict')
view_current_merge_conflict_with(context_window_size=0, reason='to view only the
current merge conflict without any local context')
view_current_merge_conflict_with(context_window_size=5, reason='to view only the
current merge conflict with some local context')
view_merge_conflict_at(conflict_index=1, context_window_size=5,
    reason='To ensure that the resolution for CONFLICT-0 is cohesive with
    CONFLICT-1')
view_merge_conflict_at(conflict_index=1, context_window_size=10,
    reason='To remind myself of the changes and context around CONFLICT-3 so
    that I can decide whether to delete the import for ShoppingClient in the
    current conflict')
view_diff_for(relative_path_from_project_root='src/app/io/FileParser.java',
    reason='view the full diff between the local and incoming changes for the
    file at path')
view_diff_for(relative_path_from_project_root='src/app/api/quantative_methods/
    regression.python', reason='understand how to resolve the current conflict such
    that the resolution is cohesive and makes sense in the context of the overall
    changes')
view_file_at(relative_path_from_project_root='src/tests/
    test_data_transformations.py', reason='understand the full context of the merge
    conflict, because I think I might have found a small bug, but I need more context
```

Figure 3: Our MCR prompt continued.

MCR Prompt - Part 3

```
    to make sure it is one before applying a minor fix as part of the conflict resolution.')
view_file_at(relative_path_from_project_root='src/app/utils.py', reason='to check
    whether there are other commented out code blocks')
view_file_at(relative_path_from_project_root='src/app/Authenticator.java',
    reason='to see how the changes I made so far fit into the file and to ensure
    I resolve the current conflicts such that it is cohesive with these previous
    resolutions')
resolve_current_merge_conflict_with(content='from app.api.auth import
    PremiumUser\\n', reason='The premium user class is a new authentication
    class that is being used in the incoming changes and thus is most likely part
    of what the junior developer wants to have access to')
resolve_current_merge_conflict_with(content='    bool debug = conf.shouldDebug;
    \\n    bool enableCaching = conf.enableCaching;\\n    bool shouldRetry =
    conf.shouldRetry;\\n', reason='both of these configuration flags are being
    used in the local changes, also I fixed a copy-paste bug and now the
    enableCaching flag is correctly initialized to conf.enableCaching. The
    shouldRetry flag is an incoming change that conflicts with what the developer
    introduced, I will thus keep all three flags.')

Key Requirements:
- Once the conflicts are resolved you are done and must terminate.
- Your decisions must be cohesive across merge conflicts.
- Make sure that all your lines end with a newline character to avoid introducing
accidental changes.
- You must always fill all parameters of the provided tools. This includes
the ”reason” parameter.
```

Figure 4: Our MCR prompt continued.

Interactive Rebase (IR) Prompt - Part 1

```
You are a staff software engineer with expertise in {programming_language} and Git.
You are helping a junior team member who has been committing all day without
pushing their commits to the remote. Help them perform an interactive rebase
to clean up their local tree. The rebase has already been initiated for you
and is currently paused so that you can inspect the commits participating in
the rebase and edit the rebase todo list.

The commits involved in the rebase are listed below. When referring to them in
function calls, use the commit index ”i” to refer to <COMMIT-i>. Avoid viewing
all commits again, they are already presented below. The commits are delimited
by the <COMMIT-i> and </COMMIT-i> tags:
{participating_commits}

Instructions:
Consider the changes in the commits and make adjustments if necessary such that the
local tree:
- contains logically cohesive commits
- all commits have meaningful, descriptive commit messages that follow a unified
format
- does not contain commits with duplicate commit messages
- follows best practices for maintainable code

You must only use the following tools and follow their specification exactly.
Always provide a reason for calling a tool.
List of valid tools for this scenario:
- view_rebase_todo: View current rebase todo list
- execute_rebase: Execute the rebase with the current rebase todo list, thereby
all rebase-todo-list-items are processed in an ascending order
```

Figure 5: Our IR prompt.

IR Prompt - Part 2

```
- show_changes_in: If you want to spend more time thinking about some of the presented commits,
use this tool to inspect the changes introduced by commit with index i. Below are some examples
of how to use this function:
    show_changes_in(commit_index=4, reason='to inspect the changes in COMMIT-4')
    show_changes_in(commit_index=0, reason='to understand how the changes in
    COMMIT-0 relate to its commit message')
- update_rebase_todo_list: Update the rebase todo list, reordering items or adjusting the
commands to perform on commits. Each item in the list that you must pass to
update_rebase_todo_list must be a string that complies with the rebase-todo-list-item
JSON schema specified below:
{{
    "type": "json",
    "schemaName": "rebase-todo-list-item",
    "schema": {{
        "type": "object",
        "properties": {{
                "commit_index": {{"type": "integer"}},
                "command": {{"enum": ["pick", "drop", "fixup",
                            "fixup -c", "squash", "reword"]}},
                "commit_msg": {{"type": "string"}},
            }}
        }},
        "required": ["operations"],
        "additionalProperties": False
}}

Below are some examples of how to use this function:
Note: Positioning the rebase todo item with index 2 at the first position in the list, will swap
it to the topmost position in the rebase todo list
    update_rebase_todo_list(rebase_todo_list_items=[
        '{{"commit_index": 2, "command": "pick"}}',
        '{{"commit_index": 1, "command": "reword", "commit_msg": "FIX:
        Explicitly handle division by zero edge case"}}',
        '{{"commit_index": 0, "command": "fixup"}}',
        '{{"commit_index": 3, "command": "pick"}}',
        '{{"commit_index": 4, "command": "drop"}}'
   ], reason='to remove an unnecessary, noise, experimental commit, improve the commit message of
    COMMIT-1 and consolidate the changes in COMMIT-0 and COMMIT-1')
Note: Example for a different sample, you must ensure to always have exactly one item per commit.
    update_rebase_todo_list(rebase_todo_list_items=[
        '{{"commit_index": 0, "command": "pick"}}',
        '{{"commit_index": 2, "command": "squash", "commit_msg": "ADD:
        Define interfaces and test cases for ShoppingBasketService"}}',
        '{{"commit_index": 1, "command": "pick"}}'
   ], reason='to reorder the local tree, yielding more coherent and logical increments of changes in
    the local tree and to consolidate the changes in COMMIT-0 and COMMIT-2')

Only the following commands are allowed for the rebase todo list items. Make sure to only
provide the required fields for each command, all fields other than the required fields are
invalid:
- pick: Use this commit as is. Required fields: ["commit_index", "command"]
- drop: Remove this commit. Required fields: ["commit_index", "command"]
- fixup: Meld this commit into previous commit, reducing the total amount of commits by 1.
Only keep the previous commit's log message. Required fields: ["commit_index", "command"]
- fixup -C: Meld this commit into previous commit, reducing the total amount of commits by 1.
Only keep this commit's log message. Required fields: ["commit_index", "command"]
- squash: Meld this commit into previous commit, reducing the total amount of commits by 1.
Commit message of resulting commit must be specified. Required fields: ["commit_index",
"command", "commit_msg"]
```

Figure 6: Our IR prompt.

IR Prompt - Part 3

```
- reword: Use commit, but edit commit message. Commit message must be specified.
Required fields: ["commit_index", "command", "commit_msg"]

Key Requirements:
- You must not simply pick all commits without modifying anything in the rebase
todo list. Do your best to improve the local tree however you see fit.
- Avoid squashing all commits into a single commit, consider for which commits this
would improve the resulting commit history.
- Try to consolidate the total size of the local tree such that the resulting tree
has length k<{times_seen_consecutively}
- You must always fill all parameters of the provided tools. This includes the
"reason" parameter.
```

Figure 7: Our IR prompt.

Iterative Committing of Changes (ICC) Prompt - Part 1

```
You are a staff software engineer with expertise in {programming_language} and Git.
You are helping a junior team member who has been working all day without creating
a commit to iteratively create commits and introduce their changes into the
repository in a maintainable way. Help them to select hunks such that you can create
multiple, small, but logically cohesive commits that are structurally sound, and
follow best practices for maintainable code.

Instructions:
- Review the remaining hunks of code and help the junior engineer select the
appropriate hunks for each commit.
- Ensure that you select as many hunks as you need to ensure structural integrity,
ie avoid breaking changes by, for example, removing a variable definition or
initialization in one commit, but removing the usages of the variable in another
commit.
- Identify the ids of the hunks that you should pass by the number following
"HUNK-" in the list of remaining hunks below. For HUNK-8, the id you need to
pass, if you want to select this hunk, would be 8.
- Each commit should be focused, small, and logically cohesive.
- Provide a clear and concise commit message for each commit following the format
provided in the example usages.

Key Requirements:
- Avoid apply all changes in a single commit unless you are absolutely sure this
will yield the best possible git history.
- You must always fill all parameters of the provided tools. This includes the
"reason" parameter.

Process all of the following {number_of_remaining_hunks} hunks:
{remaining_hunks}

Task:
Pass a list of hunks to include in the commit and a descriptive commit message
to the provided tool.

You must only use the following tools and follow their specification exactly
and always provide a reason for calling a tool.
All tools other than the ones listed below are INVALID and you MUST NOT use them
under any circumstances.
Valid tools:
- commit_changes_in
- commit_remaining_changes
```

Figure 8: Our ICC prompt.

Iterative Committing of Changes (ICC) Prompt - Part 2

```
Example usages:
    commit_changes_in(selected_hunks=[1,3], commit_message="FIX: Handle edge
        case of uninitialized object”,reason="to group the fixing of uninitialized
        objects together”)
    commit_changes_in(selected_hunks=[4], commit_message="ADD: Introduced
        new enum class CarConfiguration”, reason="to isolate the addition of the
        new enum class”)
    commit_changes_in(selected_hunks=[2,5], commit_message="REFACTOR: Migrate
        car configurator to CarConfiguration enum”, reason="The remaining changes
        both deal with migrating the existing implementation to the enum introduced
        in the previous commits. This way the commits build on each other in a
        logical progression and the migration takes place once we ensure that the
        class we migrate to is already present, thus avoiding breaking changes.”)

    Once you have received a signal that you are done, you must always call
    the tool in the example below to terminate:
    commit_remaining_changes(commit_message="UPDATE: Implement data
        streaming feature”, reason="because all hunks were processed and
        I must now terminate”)
```

Figure 9: Our ICC prompt continued.

LLM-as-a-Judge Evaluation Prompt - Part 1

```
Please act as an impartial judge and evaluate the quality of the two
git histories that are displayed below. Your evaluation should consider the
following aspects:
- The quality of the commit messages with respect to consistency, conciseness,
duplication and correctness with respect to the content of the commit.
- The logical cohesion of the changes present within the commits. Changes in
a commit should have high logical cohesion.
- The logical progression and common thread between the commits and especially
the order in which the commits are presented.
- The size of the commits. Commits should be as small as possible without
breaking the system (e.g. changing a method signature in a non-backwards
compatible way without also changing all uses of the method in the same commit).

Your job is to evaluate which git history is of higher quality. Avoid any position
biases and ensure that the order in which the responses were presented does not
influence your decision. Do not allow the length of the responses to
influence your evaluation. Be as objective as possible.

You must adhere to the response format demonstrated in example responses below:
{{
    'evaluation_result': 'HISTORY-1',
    'evaluation_reason': 'The first git history has more descriptive commit and
        non-duplicate messages that align much more accurately with the content
        of the commits.'
}}
{{
    'evaluation_result': 'HISTORY-2',
    'evaluation_reason': 'The commits in git history 2 are more concise and
        introduce logically coherent changes. The changes are introduces in such
        a way that they are unlikely to break the system as the commits are self-
        contained with respect to the part of the system that they affect and
        correctly propagate changes throughout the system. Thus I chose history
        2 despite it having poorer quality commit messages.'
}}
```

Figure 10: Our LLM-as-a-Judge evaluation prompt. We use the same prompt for both evaluation runs, we simply swap the positions of the histories that are evaluated in the prompt.

LLM-as-a-Judge Evaluation Prompt- Part 2

```
{{
    'evaluation_result': 'TIE',
    'evaluation_reason': 'Both histories introduces changes that are logically
        coherent and have similar commit messages. None of the two histories have
        fundamental issues, such as duplicate commit messages or changes that
        obviously would break the system if they were introduced as presented.
        As I am unsure, I am declaring a tie.'
}}

<HISTORY-1>
{history_1}
</HISTORY-1>
<HISTORY-2>
{history_2}
</HISTORY-2>
```

Figure 11: Our LLM-as-a-Judge evaluation prompt continued.