Merge-Bench: Resolve Merge Conflicts with Large Language Models

Benedikt Schesch[1][0009−0002−2885−3067]⋆ and Michael D. Ernst[2][0000−0001−9379−277X]

[1] Amazon scheschb@amazon.co.uk
[2] University of Washington, USA mernst@cs.washington.edu

Abstract. This paper applies machine learning to the difficult and important task of version control merging. (1) We constructed a dataset, Merge-Bench, of 7938 real-world merge conflict hunks from 1439 GitHub repositories. The ground truth is the merge resolution that developers committed to the repository. Our dataset construction methodology is scalable to arbitrary amounts of data since no manual labeling is required. (2) We trained a model, LLMergeJ, to resolve merge conflicts in Java programs. Our approach uses Group Relative Policy Optimization (GRPO), an online reinforcement learning method, to train a Large Language Model (LLM). (3) We performed two evaluations of the performance of LLMs on resolving merge conflicts. On Java programs, LLMergeJ with 14B parameters outperforms 3 commercial LLMs, trailing only Gemini 2.5 Pro. Across 11 programming languages, commercial LLM performance is largely stable from language to language. The best models correctly resolve less than 60% of merge conflicts.

Keywords: Benchmark · Dataset · Version Control · Merging.

1 Introduction

A merge conflict occurs when a version control system cannot automatically integrate concurrent changes from different developers. A merge conflict requires manual resolution, consuming significant developer time and often causing software defects [2]. Traditional automated merge tools rely on syntactic and structural heuristics, but they struggle with semantic conflicts that require understanding code intent and behavior. Large Language Models (LLMs) offer a promising approach to automate merge conflict resolution by leveraging their code understanding capabilities [3, 24].

Training and evaluating a merge tool requires a dataset of problems and ground-truth solutions. We created a new dataset, **Merge-Bench**, containing 7938 real-world merge conflict hunks from 1439 public GitHub repositories. The

⋆ Work done independently of the author's role at Amazon.

ground truth is the programmer's merge resolution committed to the repository. Our code and data are publicly available.[3]

Merge-Bench does not suffer from common, serious problems in other benchmarks (further discussed in the related work section). It is **realistic** because the merge conflicts come from real-world, widely used repositories. It suffers little **data leakage** because the textual conflicts used in its problem statements do not appear explicitly in the GitHub repository. (However, the merge resolutions (ground truth) and surrounding context lines are almost certainly in commercial LLM training data.) It is **scalable** because neither dataset creation nor merge tool evaluation requires human effort, such as creating ground truth. It is not prone to **reward hacking**, where the model optimizes the reward function to the detriment of real-world usefulness. Reward hacking is a significant problem with evaluations that run a program's existing test suite. We introduce a test-free paradigm for evaluating merge conflict resolution, along with a normalization procedure for the output of merging tools.

We developed **LLMergeJ**, a 14B parameter model trained with Group Relative Policy Optimization (GRPO) [21] to resolve merge conflicts in Java. LLMergeJ demonstrates the use of Merge-Bench, illustrates the weaknesses of commercial LLMs on the task of merging, and shows the strength of reinforcement learning.

Our contributions include:

- **Test-free evaluation paradigm for evaluating merges**: A methodology that bypasses test execution, enabling scalable training on millions of resolved conflicts without reward-hacking vulnerabilities.
- **Human-filter-free SWE-RL task**: A software-engineering-oriented reinforcement learning task that combines internet-scale data with zero human filtering or manual adjustments, enabling truly scalable training.
- **Merge-Bench**: A scalable benchmark resistant to reward hacking and requiring no manual annotation.
- **LLMergeJ**: The first model trained with online RL for merge conflict resolution, achieving 49% exact match accuracy and 59% source code match accuracy on real-world conflicts.
- **Evaluation of LLMergeJ**: Experiments showing that our 14B parameter model achieves competitive or better performance than 3 commercial LLMs.
- **Cross-language evaluation**: Experiments showing the performance of 6 state-of-the-art commercial models across 11 programming languages.

2 Related Work

2.1 Software engineering task benchmarks

Previous LLM evaluations for code rely on two primary approaches: real code or interview questions. Each has fundamental limitations.

[3] Dataset construction: `https://github.com/benedikt-schesch/Merge-Bench-Builder`, Model training: `https://github.com/benedikt-schesch/LLMerge`, Evaluation: `https://github.com/benedikt-schesch/Merge-Bench`

Real programs. SWE-Bench [9] validates model outputs by executing project test suites. This approach suffers from a critical vulnerability — models learn to satisfy tests rather than produce correct code. This reward-hacking problem is not hypothetical. GenProg [26] demonstrated that evolutionary algorithms could generate program repairs (patches) satisfying test suites. However, these patches *usually* made the tests pass by deleting functionality or tests or by overfitting to specific inputs [15]. LLMs, too, may "solve" programming tasks by exploiting memory vulnerabilities to avoid correctness checks [18].

SWE-Bench has similar problems. Later researchers [14] found that SWE-Bench systematically underestimated AI model capabilities due to faults in SWE-Bench itself: unreliable development environments causing false test failures, problem descriptions too vague for AI systems to understand properly, and overly restrictive unit tests requiring exact matches for undocumented implementation details. To address these concerns, SWE-Bench Verified [14] employed 93 professional software developers for manual annotation and developed a Docker-based evaluation harness, reducing the dataset by 78% from 2,294 to 500 verified samples. This costly process is not scalable.

As a point of comparison, Merge-Bench's 1439 repositories and 7938 samples is much larger than SWE-Bench's 12 repositories and 2294 or 500 samples. Merge-Bench is also easy to expand in size, with no human judgment required.

Interview questions. LiveCodeBench [8] consists of algorithmic problems from interview platforms like LeetCode. A positive is that these benchmarks avoid some reward-hacking issues through their simplicity and robust test suites. However, they are fundamentally unrealistic and do not represent the complexity of real repositories where developers must navigate existing code, understand context, and maintain compatibility with surrounding systems.

2.2 Merging benchmarks

ConflictBench [22] contains 180 hunks, which the authors call "merging scenarios". ConflictBench contains only hunks where each version has no more than 20 lines of text. (Our benchmark Merge-Bench has the same size restriction.)

GitGoodBench [12] contains about 337 merge resolution problems. The authors' filtering discarded some merges, including those with more than 8 hunks, those with conflicts in non-code files, etc. ConGra [29] mined 44,948 conflicts from 34 projects across 4 programming languages. While ConGra is larger than Merge-Bench in raw conflict count, it has less diversity in terms of repositories and programming languages. The dataset's size makes evaluation on commercial LLMs prohibitively expensive (at least tens of thousands of dollars). Consequently, ConGra's evaluation focused primarily on smaller open-source models, with Llama3 8B performing best. Its evaluation protocol instructs LLMs to always attempt resolution rather than preserving conflicts when developer intent is ambiguous. Furthermore, ConGra considers a resolution correct if any similarity metric exceeds 80%, which does not guarantee semantic equivalence and may accept functionally incorrect solutions.

2.3 LLM-based merging tools

MergeBERT [24] converts the merge problem into a classification problem. It neurally classifies a merge into one of 9 resolution patterns, such "choose the left text", "choose the right", "choose the base", "concatenate the left and right", etc. The authors claim 63–68% accuracy. MergeBERT is not publicly available. DeepMerge [3] outputs a proposed merge, where each line of the merge is a line from one of the two versions being merged. DeepMerge's precision is less than that of jsFSTMerge [25], a semi-structured merge tool that uses source text and ASTs. DeepMerge is not publicly available.

MergeGen [4] treats the merge problem as a generation problem rather than a classification problem (as MergeBERT and DeepMerge did). The MergeGen authors trained an encoder and decoder and obtained results slightly better than those of MergeBERT.

Gmerge [28] is a wrapper around GPT-3, where the prompt engineering embodies k-shot learning by incorporating several examples in the prompt. Gmerge is not publicly available. ChatMerge [23] trains a classifier; if the classifier is unsuccessful, then ChatMerge calls out to ChatGPT to get an answer.

3 Merge-Bench

3.1 Structure of the benchmark

In a version control system, a *merge* integrates two versions of text, where each version was created by independent, concurrent work. The two versions of text are called *left* and *right*; *base* is their common ancestor, which both versions edited. A version control system can automatically integrate non-overlapping edits; however, if both versions edit the same part of the base document, then the version control system presents merge *conflicts* to the user. A conflict consists of a set of *hunks*, each of which represents a minimal part of a document that both versions edited differently (plus a few nearby context lines that are identical in base, left, and right). In each hunk, the left and right text differ. The textual representation of a hunk is a *3-way diff* that includes the left, base, and right text separated by conflict markers such as `<<<<<<<`, `=======`, and `>>>>>>>`.

The **Merge-Bench benchmark** consists of a set of merge conflict hunks. The merge conflict hunk contains text for left, right, and base. It also contains pre-context and post-context, the common text before and after the hunk. Context is important because resolving a conflict typically requires examining the surrounding code. Merge-Bench represents a conflict exactly as git and diff3 do: as a 3-way diff with context lines before and after it.

For each merge conflict hunk, Merge-Bench also contains the ground truth: the text that the programmer manually committed to resolve the conflict.

3.2 Dataset construction

Public version control repositories are a rich source of merge conflicts. A version control repository contains a history of merge conflicts and how programmers resolved them.

Candidate repository selection. We began with a large pool of candidate repositories to ensure sufficient diversity and coverage across programming languages. For most languages, we used the Reaper dataset [13], which contains high-quality projects from the GHTorrent dataset [5]. Go, JavaScript, TypeScript, and Rust are not present in the Reaper dataset; for them we used the top 1000 most-starred repositories from GitHub for each language. For Java specifically, we reserved the first 1000 most-starred repositories as candidates for training; the following 200 repositories are candidates.

Merge collection. From each candidate repository, we systematically collected merge conflicts by analyzing up to 1000 branches per repository, including the main branch, feature branches, and deleted branches (which remain accessible through GitHub's API). Merges from non-main branches are particularly valuable because they tend to be more challenging than merges on the main branch [19]. We replayed each merge using git to obtain conflicts. We sampled hunks from each repository to obtain 600–800 hunks per language. We kept the number of sampled merges per repository low to increase dataset diversity.

Many candidate repositories did not contribute to the final dataset due to two primary factors: (1) repositories that became inaccessible or were deleted after our initial selection (due to its age, Reaper now contains many deleted repositories), and (2) repositories that contained no merge conflicts in our sampled branches or whose conflicts were filtered out during filtering.

Resolution extraction and filtering. Each hunk in Merge-Bench is computed using a context size of 20 lines.

Merge-Bench is a set of conflicting hunks, each of which a merge tool analyzes independently. We obtained the merge resolution of each hunk by setting the context size up to 20 lines before and after each conflict, but we prevented it from spanning multiple conflicts. To prevent misidentifying the resolution from the merged code, we discarded hunks with missing or repeated context lines in the ground truth. We also discarded hunks whose resolution is larger than the sum of left, base, and right, since such hunks likely contain new code beyond what was in the files being merged. Due to our limited computational resources, we enforced size constraints: we removed hunks where the left, right, base, or ground truth text exceeds 20 lines, or where the merge conflict itself exceeds 512 tokens (using the DeepSeek R1 tokenizer). This conservative filtering ensures high-quality data while maintaining full automation, requiring no manual verification or annotation.

The algorithms that build Merge-Bench can be run on more repositories and with different parameters, including providing complete files or complete repositories to the LLM.

Final dataset composition. The final dataset (fig. 1) contains 7938 hunks across 11 programming languages, with an average of 5.5 hunks per repository.

Language	C	C++	C#	Go	Java	Java Script	PHP	Python	Ruby	Rust	Type Script	Total
# **hunks**	630	787	777	676	806	759	574	761	653	716	799	7938
# **repos**	62	150	172	87	44	157	109	184	157	147	170	1439

Fig. 1. Size of the Merge-Bench dataset, by language.

4 Code Comparison Methods

When a tool produces code, the generated code must be compared against the ground truth code. Here are ways to do the comparison.

Textual. The generated code is considered correct if it exactly matches the ground truth. This approach is scalable and language-agnostic. However, it is overly restrictive and underestimates the quality of the generated code. It rejects semantically equivalent code that differs only in formatting, comments, etc.

Source code. The generated code and ground truth are normalized by comment removal and standardized formatting. This is similar to comparing ASTs (abstract syntax trees or parse trees). Requiring only a parser per programming language, this approach is scalable and less restrictive than textual comparison. However, it still rejects some semantically equivalent code that uses different variable names or is structurally different. In other words, it is a proxy for, and lower bound on, correctness.

Semantic. The program equivalence problem (determining whether two programs have identical behavior) is undecidable [17]. Approximating it requires sophisticated, non-scalable program verification. It requires access to all libraries used in the program, which can be difficult for real-world software.

Testing. A test suite can determine that two programs behave the same on a specific finite set of outputs, according to a finite number of assertions. Testing accepts semantically equivalent programs that are syntactically different. A negative is that it accepts semantically *different* programs that happen to behave the same for certain inputs. Furthermore, testing can be computationally expensive. Automated testing requires a set of programs to use a standardized build and test system for all their tests. Flaky tests that fail non-deterministically can yield misleading results. Most importantly, testing requires a test suite with high coverage and strong assertions that an LLM cannot reward-hack.

Although semantic equivalence would be the best comparison method, it is infeasible. Therefore, we use source code equivalence as a practical and scalable approximation that captures many cases of functionally equivalent code. Our normalization procedure removes block comments (e.g., `/* ... */`), line comments (e.g., `// ...`), and docstrings (e.g., `"""..."""`), using language-specific rules. Our normalization procedure also standardizes whitespace. It removes leading whitespace (except for whitespace-sensitive languages like Python and Ruby) and trailing whitespace, and it collapses all consecutive whitespace characters to a single space.

You are a merge conflict resolution expert. Below is a snippet of code with surrounding context that includes a merge conflict.
Return the entire snippet (including full context) in Markdown code syntax as provided.
Do not modify the context at all and preserve the spacing as is.
Think in terms of intent and semantics that both sides of the merge are trying to achieve.
If you are not sure on how to resolve the conflict or if the intent is ambiguous, please return the same snippet with the conflict.
Here is the code snippet:
```<language>
<conflict>
```

Fig. 2. User prompt for merge conflict resolution.

5 LLMergeJ Training

We trained LLMergeJ, a small but capable LLM for resolving Java merge conflicts. Our experiments (section 7) evaluated LLMergeJ against commercial LLMs, revealing the strengths and weaknesses of general-purpose commercial LLMs for resolving merge conflicts.

Due to resource constraints, we trained LLMergeJ focused on one programming language. The methodology readily scales to millions of samples given the vast repository of merge conflicts available in public repositories, because our approach requires no manual verification or annotation.

For training, we used the procedure of section 3.2 with the first 1000 starred Java repositories, which are distinct from the Merge-Bench repositories. This training set has 67 Java repositories and 3424 selected conflict hunks.

Model Input. We used DeepSeek R1's system prompt and the query in fig. 2, adding the conflicting code and language marker.

Reward Structure. LLMergeJ's reward function is the sum of three components $R_{\text{total}} = R_{\text{reasoning}} + R_{\text{format}} + R_{\text{resolution}}$ where:

$$R_{\text{reasoning}} = 1 \text{ if output uses } \texttt{<think>} \text{ tokens}, 0 \text{ otherwise}$$
$$R_{\text{format}} = 1 \text{ if output has correct Markdown formatting}, 0 \text{ otherwise}$$
$$R_{\text{resolution}} = \begin{cases} 1.0 & \text{if textual match with ground truth} \\ 0.5 & \text{if source code match (normalized)} \\ 0.1 & \text{if conflict preserved (no resolution)} \\ 0 & \text{otherwise (likely incorrect merge)} \end{cases}$$

GRPO. We trained our model using GRPO [6,21], which is a simpler policy optimization approach than PPO [20] since it does not require the construction of a critic network. GRPO works by sampling a few rollouts from a network and reinforcing the higher-reward ones relative to the others. To estimate the advantage, it uses the mean reward of all rollouts, which replaces the usual critic

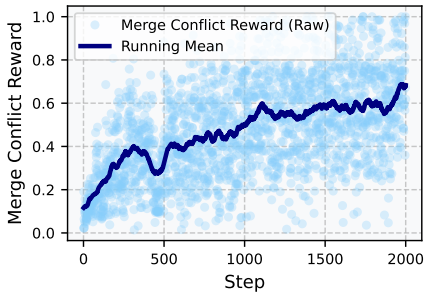

Fig. 3. Merge conflict reward over time during the training process.

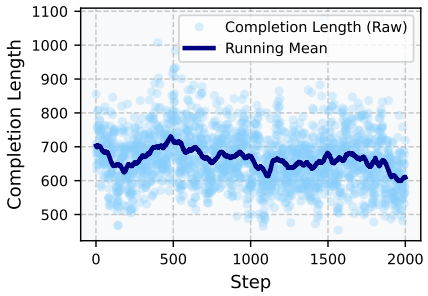

Fig. 4. The evolution of completion length over the training process.

baseline with a group-normalized return. For each prompt q we

$$\begin{aligned} \text{sample } G \text{ outputs} \quad & \{o_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q) \ , \\ \text{compute standardized advantages} \quad & A_i \ = \ \frac{r_i \ - \ \text{mean}\big(\{r_1, r_2, \ldots, r_G\}\big)}{\text{std}\big(\{r_1, r_2, \ldots, r_G\}\big)} \ , \\ \text{and define the probability ratio} \quad & \rho_i(\theta) = \frac{\pi_\theta(o_i \mid q)}{\pi_{\theta_{\text{old}}}(o_i \mid q)} \ . \end{aligned}$$

The objective function of GRPO is then expressed as

$$J_{\text{GRPO}}(\theta) = \mathbb{E}_{q,\{o_i\}}\Bigg[\frac{1}{G}\sum_{i=1}^{G} \min\big(\rho_i(\theta)\, A_i,\ \text{clip}(\rho_i(\theta),\, 1-\epsilon,\, 1+\epsilon)\, A_i\big) \\ - \ \beta\, D_{\text{KL}}\big(\pi_\theta(\cdot \mid q) \ \| \ \pi_{\theta_{\text{old}}}(\cdot \mid q)\big)\Bigg] \ .$$

GRPO over Supervised Fine-Tuning. To demonstrate the effectiveness of our reinforcement learning approach, we compare GRPO against Supervised Fine-Tuning (SFT). SFT faces two fundamental challenges in merge conflict resolution: the inability to optimize reasoning processes and the unpredictability of ground truth solutions.

The most significant advantage of GRPO is its ability to fine-tune the reasoning step. Our reward structure includes the traditional reasoning reward, allowing GRPO to optimize not just the final merge resolution but also the intermediate reasoning process. SFT, in contrast, can only learn from the final output: it cannot improve the model's reasoning capabilities during conflict analysis.

Additionally, developers may resolve two similar conflict hunks in semantically equivalent but syntactically different ways. For instance, a developer might choose to refactor the conflicting code, using a cleaner implementation that achieves the same functionality. SFT would create misleading gradients, potentially degrading the model's ability to generate alternative valid solutions.

GRPO addresses these challenges through its group-relative optimization approach. When the ground truth is unpredictable due to multiple valid resolutions, most model outputs will receive a reward of 0 (for incorrect attempts), while outputs that preserve the conflict markers receive a small positive reward of 0.1. The group averaging then creates gradients that favor conflict preservation over incorrect resolution attempts. It does not favor any one resolution over the others, while still allowing the model to learn from cases where the resolution is more predictable. Combined with the reasoning reward, this approach enables the model to develop better analytical capabilities while avoiding overfitting to specific syntactic patterns.

Training Procedure. We fine-tuned the DeepSeek-R1 14B distilled Qwen model, which has been quantized by unsloth [7] to 4 bits. This model already has reasoning incorporated into it. To reduce training requirements we use lora adapters with rank and alpha value of 128; this allows us to not train the entire network. We additionally limit the input prompt length to 512 tokens and the output length to 2048 tokens. We set the model temperature to 0.9 and the number of generations per sample to 16 with a batch size of 1 and gradient accumulation steps set to 4. As optimizer we use the paged adamw with $\beta_1 = 0.9$, $\beta_2 = 0.99$, weight decay 0.0, and learning rate 5e-5 with constant warmup over 30 steps. We use standard GRPO parameters: clipping $\epsilon = 0.2$, KL coefficient $\beta = 0.0$ (no reference model for memory efficiency), gradient clipping at max norm 0.2, and DAPO loss normalization. The training process took 4 days on an H200 GPU with AMD EPYC 9534 64-Core Processor, 1.5TB memory, running Red Hat Enterprise Linux 9.5. Figure 3 shows how the model's merge conflict reward improves during training. The detailed plots of all reward components during training are provided in the technical appendix.

Figure 4 shows how the completion length (which includes `<think>` tokens) changes during training. Notably, this graph remains relatively stable, unlike general-purpose models such as R1, where completion length increases during training [6]. This stability likely reflects our constrained experimental setup with limited token budgets and shorter training duration.

6 LLM models evaluated

Figure 5 shows the LLMs that we evaluated on the task of resolving Java conflicts, all prompted zero-shot with the query from fig. 2 and no in-context examples. Our procedure can be applied to any programming language, but our small LLMergeJ 14B-parameter model has only been trained on Java. Java is most common in prior research in merge conflict resolution [1, 10, 11, 19].

Our model has 14B parameters. We also trained a supervised fine-tuning (SFT) baseline using Qwen3-14B [27]. We intentionally use different base models for our comparison: LLMergeJ builds on DeepSeek-R1 (a reasoning-capable model) while the SFT baseline uses Qwen3-14B (a more recent and capable standard model). This design choice is necessary because SFT cannot optimize reasoning processes (unless SFT is run on reasoning traces including <think> to-

kens) — it can only learn from final outputs. Applying SFT to a reasoning model would fail to leverage its reasoning capabilities, as SFT has no mechanism to improve the intermediate reasoning steps that reasoning models generate. By using the more recent Qwen3-14B for SFT, we ensure a fair comparison that, if anything, favors the SFT baseline. This allows us to evaluate whether GRPO's ability to optimize reasoning provides advantages over traditional SFT even when SFT uses a stronger base model. The SFT baseline's training set is the same as LLMergeJ and contains the ground truth developer solutions from 3424 merge conflicts as its target. We conducted an extensive hyperparameter search using Qwen3-14B as the base model, systematically evaluating 24 different configurations across learning rates (1e-4, 1e-5, 1e-6), weight decay values (0, 0.01), learning rate schedulers (linear, cosine), and training epochs (1, 3). In fig. 5, "Qwen3-14B SFT" represents the optimal configuration from this comprehensive search, selected based on the highest percentage of correctly resolved merges on the test set of 806 conflicts. The detailed results of each configuration are provided in the technical appendix. This process heavily favors SFT over GRPO. Distillation and SFT models also receive 4× higher token limits (8,192 vs. 2,048 output tokens).

7 Results and Analysis

Our experimental infrastructure classifies a merge resolution into five categories:

Equivalent Text: The model's output exactly matches the ground truth resolution as a string, including all whitespace, comments, and formatting.

Code Normalized Equivalent: The model's output differs from the ground truth only in formatting or comments. Note that Textual Match is a subset of Normalized Source Code Match.

Conflict: The model outputs the original conflict markers unchanged, indicating uncertainty about the correct resolution. This is considered a valid response when the merge intent is ambiguous, as instructed in our prompt.

Different Code: The model's normalized output differs from the normalized ground truth. This represents possibly incorrect resolution attempts.

Invalid Markdown: The model's output does not contain a Markdown fenced code block (missing or malformed ‘‘‘ markers).

The last four categories are mutually exclusive and collectively exhaustive.

7.1 Java Experiments

Remarkably, our 14B parameter LLMergeJ model is competitive with the largest available models at the time of our experiments. Our model ranks second in textual and source code equality (fig. 5). Gemini 2.5 Pro is best. LLMergeJ outperforms commercial models like Claude Opus 4, o3 Pro, and Grok 4 that have orders of magnitude more parameters. This represents a substantial parameter efficiency gain.

Model	Comparison to developer				
	Equivalent text	Code normalized equivalent	Different code	Conflict (no resolution)	Invalid Markdown
Gemini 2.5 Pro	**54.7%**	**62.5%**	34.1%	3.4%	0.0%
o3 Pro	46.1%	54.3%	35.1%	10.6%	0.0%
Claude Opus 4	44.4%	51.2%	27.6%	21.2%	0.0%
Grok 4	33.4%	39.7%	17.9%	42.4%	0.0%
Llama 4 Maverick	26.2%	32.6%	27.1%	40.1%	0.2%
QwQ 32B	32.1%	43.2%	30.9%	20.5%	5.5%
Qwen3 8B	5.5%	9.1%	4.7%	86.1%	0.1%
Qwen3 14B	12.9%	16.6%	8.7%	74.7%	0.0%
Qwen3 32B	13.2%	16.9%	11.1%	71.8%	0.1%
Qwen3 235B	30.9%	39.5%	25.4%	35.1%	0.0%
R1 1.5B	0.0%	0.2%	46.5%	44.0%	9.3%
R1 8B	3.5%	8.1%	31.6%	58.4%	1.9%
R1 14B	9.3%	13.4%	15.4%	70.7%	0.5%
R1 32B	22.8%	30.4%	29.3%	39.7%	0.6%
R1 70B	25.7%	33.0%	26.9%	39.6%	0.5%
R1-0528 671B	35.9%	42.4%	24.0%	33.2%	0.4%
Qwen3-14B SFT	36.6%	44.7%	36.1%	19.2%	0.0%
LLMergeJ 14B	<u>48.8%</u>	<u>58.9%</u>	35.5%	5.6%	0.0%

Fig. 5. Success in merging Java conflicts. "Equivalent text" is a subset of "code normalized equivalent. The last 4 columns sum to 100%. Best results are shown in **bold** (1st place) and <u>underlined</u> (2nd place). Proprietary and public models are separated.

The SFT baseline uses hyperparameters chosen to perform best on the exact test data used for evaluation. Despite this bias in its favor, the SFT baseline outputs the same code as the developer only about $\frac{3}{4}$ as often as our GRPO-trained model. This demonstrates the benefit of reinforcement learning for merge conflict resolution.

Correct merge resolutions are not the only important metric; a model should be penalized for incorrect merge resolutions, which are worse for the programmer than leaving the conflict in place [19]. If a bad resolution is twice as bad as a correct resolution is good, then all of the models provide negative benefit to a programmer. It is straightforward to make a model less aggressive (leaving the conflict in place more often) by changing the training reward (currently 0.1 for leaving the conflict in place, as noted in section 5) for LLMergeJ and SFT, or by changing the prompts for the other models.

Our evaluation reveals three behavioral patterns among the evaluated models, providing insights into different approaches to merge conflict resolution.

– **Format-struggling models** (QwQ 32B, R1 1.5B) exhibit significant instruction-following deficits, producing 5.5–9.3% invalid outputs that fail to meet basic formatting requirements, indicating fundamental challenges in

Model	Equivalent text	Code normalized equivalent	Different code	Conflict (no resolution)	Invalid Markdown
Gemini 2.5 Pro	**47.1%**	**52.6%**	42.1%	5.3%	0.0%
o3 Pro	39.2%	45.1%	40.9%	14.1%	0.0%
Claude Opus 4	40.3%	44.8%	34.5%	20.4%	0.3%
Grok 4	27.7%	31.7%	20.9%	47.3%	0.1%
Qwen3 235B	25.8%	30.6%	32.0%	37.3%	0.1%
R1-0528 671B	32.0%	36.5%	26.3%	36.9%	0.4%

Fig. 6. Model performance summary across all languages. Columns are as in fig. 5.

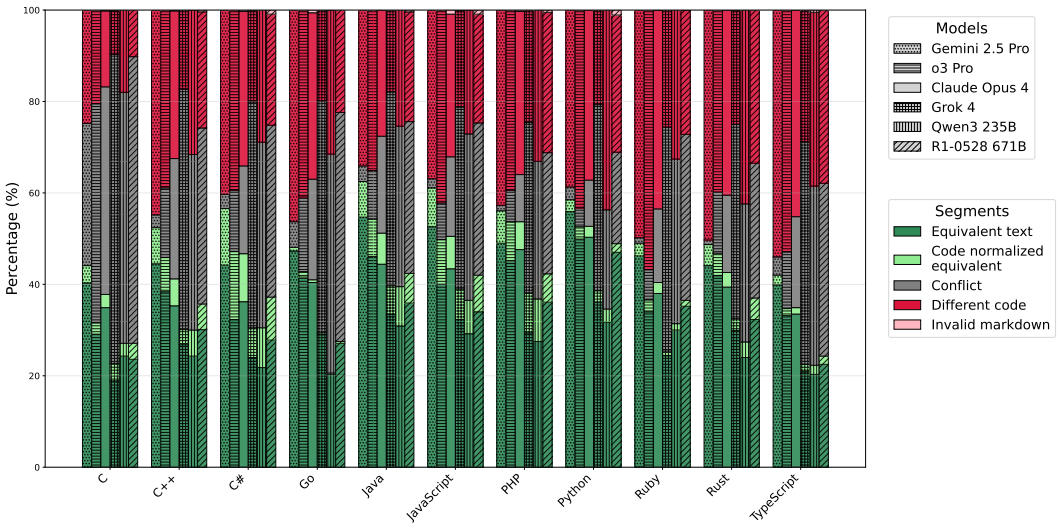

Fig. 7. Performance comparison across models and programming languages.

following the task specification. Surprisingly, this does not prevent the QwQ 32B [16] model from being competitive.

– **Conservative resolvers** (such as Grok 4, Llama 4 Maverick, Qwen3 8B) preserve 40.1–86.1% of conflicts unresolved, suggesting a cautious approach when facing ambiguous merge scenarios. Outputting a conflict does no harm, but their high conflict rates indicate uncertainty in resolution capabilities.

– **Aggressive resolvers** (including our model LLMergeJ, Gemini 2.5 Pro, o3 Pro, and Claude Opus 4) demonstrate high confidence in their resolution capabilities, successfully resolving 51.2–62.5% of conflicts with code normalized equivalent or better matches while preserving only 3.4–21.2% of conflicts unresolved. These models exhibit excellent Markdown formatting following but very rarely maintain conflicts.

7.2 Multi-language experiment

The above experiment focused on resolving merge conflicts in Java programs. Resource constraints prevented us from training LLMergeJ on all languages. Therefore, we do not know whether our results would generalize to other programming languages. To learn whether LLM-assisted merging is different in other languages, we ran the best models against the non-Java parts of Merge-Bench.

Figure 7 shows that some languages, such as C, are harder to merge; in fact, Java is one of the easiest languages to merge, along with PHP and Python. Notably, for C language conflicts, we observe that models appear to be aware of the inherent challenges and adopt extremely careful merging strategies, preserving many conflicts rather than attempting potentially incorrect resolutions. More importantly, the general patterns of LLM behavior, such as which LLMs are better than others and which LLMs give up without resolving a conflict, are *similar* across all programming languages. This is suggestive that our technique may generalize to other languages.

Figure 6 summarizes the aggregate performance. Gemini 2.5 Pro performs 7 percentage points better than the next-best models, o3 Pro and Claude Opus 4.

The evaluation on commercial LLMs cost thousands of US dollars and weeks of time. However, running our small model LLMergeJ was fast and cheap.

8 Limitations

Like any research, ours has limitations. The programmer merge might have been wrong. Although the prompt (a merge conflict) does not textually appear on the Internet, we do not know how commercial training datasets are created and they could conceivably include textual representations of merge conflicts. It is possible to choose only merge conflicts created after a model's training cutoff date. While the number of merges on GitHub is vast, it is best to select from high-quality repositories (as Merge-Bench does). We have not directly evaluated our training approach on languages other than Java. The limitation in hunk size might exclude some of the hardest merge problems, making our results an over-estimate of performance on all merge problems.

9 Conclusion

This paper presents two major contributions to automated merge conflict resolution: Merge-Bench and LLMergeJ.

Merge-Bench is a large-scale benchmark containing 7938 real-world merge conflicts from 1439 repositories. More importantly, the Merge-Bench construction methodology is scalable to the millions of repositories on GitHub. Merge-Bench addresses fundamental limitations in existing evaluation approaches. It uses real-world data, it minimizes data leakage, it is scalable (zero human labeling), and our test-free evaluation paradigm avoids reward hacking vulnerabilities.

LLMergeJ is the first approach to leverage online reinforcement learning for merge conflict resolution. Evaluation on Merge-Bench, demonstrated that our 14B parameter model performs competitively with much larger commercial models while maintaining excellent parameter efficiency. Our methodology addresses fundamental scalability limitations in existing approaches by eliminating the need for human filtering or manual adjustments, enabling truly scalable training on internet-scale data with no human annotation requirements.

Acknowledgements This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590132.

References

1. Apel, S., Liebig, J., Brandl, B., Lengauer, C., Kästner, C.: Semistructured merge: Rethinking merge in revision control systems. In: ESEC/FSE. pp. 190–200 (Sep 2011)
2. Brindescu, C., Ahmed, I., Jensen, C., Sarma, A.: An empirical investigation into merge conflicts and their effect on software quality. Empirical Softw. Engg. **25**, 562–590 (Jan 2020). https://doi.org/10.1007/s10664-019-09735-4
3. Dinella, E., Mytkowicz, T., Svyatkovskiy, A., Bird, C., Naik, M., Lahiri, S.: DeepMerge: Learning to merge programs. IEEE TSE **49**(4), 1599–1614 (Apr 2023)
4. Dong, J., Zhu, Q., Sun, Z., Lou, Y., Hao, D.: Merge conflict resolution: Classification or generation? In: ASE. pp. 1652–1663 (Sep 2023)
5. Gousios, G.: The GHTorrent dataset and tool suite. In: MSR. pp. 233–236 (May 2013). https://doi.org/https://doi.org/10.1109/MSR.2013.6624034
6. Guo, D., Yang, D., Zhang, H., Song, J., Wang, P., Zhu, Q., Xu, R., Zhang, R., Ma, S., et al.: DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. Nature **645**, 633–638 (Sep 2025). https://doi.org/10.1038/s41586-025-09422-z
7. Han, D., Han, M., Unsloth Team: Unsloth (2023), `http://github.com/unslothai/unsloth`
8. Jain, N., Han, K., Gu, A., Li, W.D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., Stoica, I.: LiveCodeBench: Holistic and contamination free evaluation of large language models for code. arXiv preprint arXiv:2403.07974 (June 2024)
9. Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., Narasimhan, K.: SWE-bench: Can language models resolve real-world GitHub issues? `{https://arxiv.org/abs/2310.06770}` (Nov 2024)
10. Larsén, S., Falleri, J.R., Baudry, B., Monperrus, M.: Spork: Structured merge for Java with formatting preservation. IEEE TSE **49**(01), 64–83 (Jan 2023)
11. Leßenich, O., Apel, S., Lengauer, C.: Balancing precision and performance in structured merge. ASE **22**(3), 367–397 (May 2014)
12. Lindenbauer, T., Bogomolov, E., Zharov, Y.: GitGoodBench: A novel benchmark for evaluating agentic performance on Git. https://arxiv.org/abs/2505.22583 (May 2025)
13. Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M.: Curating GitHub for engineered software projects. Empirical Softw. Engg. **22**(6), 3219–3253 (Dec 2017)
14. OpenAI Preparedness, NLP, P.: Introducing SWE-bench Verified. `https://openai.com/index/introducing-swe-bench-verified/` (August 2024)

15. Qi, Z., Long, F., Achour, S., Rinard, M.: An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: ISSTA. pp. 24–36 (July 2015). https://doi.org/10.1145/2771783.2771791
16. Qwen Team: QwQ-32B: Embracing the power of reinforcement learning (March 2025), https://qwenlm.github.io/blog/qwq-32b/
17. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society **74**(2), 358–366 (1953)
18. Sakana AI: The AI CUDA engineer: Agentic CUDA kernel discovery, optimization and composition. https://sakana.ai/ai-cuda-engineer/#limitations-and-bloopers (Feb 2025)
19. Schesch, B., Featherman, R., Yang, K.J., Roberts, B.R., Ernst, M.D.: Evaluation of version control merge tools. In: ASE. pp. 831–843 (Oct 2024)
20. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. CoRR **abs/1707.06347** (2017), http://arxiv.org/abs/1707.06347
21. Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y.K., Wu, Y., Guo, D.: DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. https://arxiv.org/abs/2402.03300 (apr 2024)
22. Shen, B., Meng, N.: ConflictBench: A benchmark to evaluate software merge tools. J. Sys. Softw. **214** (2024)
23. Shen, C., Yang, W., Pan, M., Zhou, Y.: Git merge conflict resolution leveraging strategy classification and LLM. In: 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS). pp. 228–239 (2023)
24. Svyatkovskiy, A., Fakhoury, S., Ghorbani, N., Mytkowicz, T., Dinella, E., Bird, C., Jang, J., Sundaresan, N., Lahiri, S.K.: Program merge conflict resolution via neural transformers. In: ESEC/FSE. pp. 822–833 (Nov 2022)
25. Trindade Tavares, A., Borba, P., Cavalcanti, G., Soares, S.: Semistructured merge in JavaScript systems. In: ASE. pp. 1014–1025 (Sep 2019)
26. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: ICSE. pp. 364–374 (May 2009)
27. Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., et al.: Qwen3 technical report. https://arxiv.org/abs/2505.09388 (May 2025)
28. Zhang, J., Mytkowicz, T., Kaufman, M., Piskac, R., Lahiri, S.K.: Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper). In: ISSTA. p. 77–88 (July 2022)
29. Zhang, Q., Su, L., Ye, K., Qian, C.: ConGra: Benchmarking automatic conflict resolution. https://arxiv.org/abs/2409.14121 (Sep 2024)