WarriorCoder: Learning from Expert Battles to Augment Code Large Language Models

Anonymous ACL submission

Abstract

Despite recent progress achieved by code large language models (LLMs), their remarkable abilities are largely dependent on fine-tuning on the high-quality data, posing challenges for data collection and annotation. To address this, current methods often design various data flywheels to collect complex code instructions, enabling models to handle more intricate tasks. However, these approaches typically rely on off-the-shelf datasets and data augmentation from a limited set of proprietary LLMs (e.g., Claude, GPT4, and so on), which restricts the diversity of the constructed data and makes it prone to systemic biases. In this paper, we propose **WarriorCoder**, a novel paradigm learns from expert battles to address these limitations. Specifically, we create an arena where leading expert code LLMs challenge each other, with evaluations conducted by impartial judges. This competitive framework generates novel training data from scratch, leveraging the strengths of all participants. Experimental results show that **WarriorCoder** achieves state-of-the-art performance compared to previous models of the same size, even without relying on proprietary LLMs.

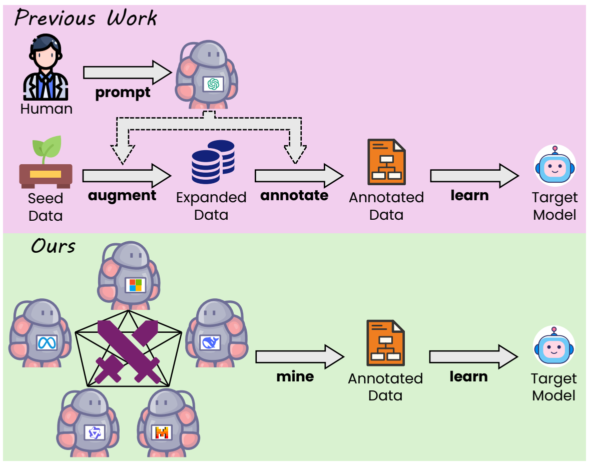

Figure 1: The comparisons between our method and traditional data flywheels. Unlike previous work, we guides the target model to learn from pairwise competitions. No demand for seed datasets, human-generated prompts, or annotations from proprietary models, the target model integrates the strengths of its competitors.

1 Introduction

Recent large language models (LLMs) have demonstrated impressive performance on code-related tasks (Li et al., 2023; Rozière et al., 2023; Guo et al., 2024; DeepSeek-AI et al., 2024; Li et al., 2022; Nijkamp et al., 2023; Zheng et al., 2023b; Fried et al., 2023; Wang et al., 2021). These successes highlight that pre-training on vast amounts of code data significantly enhances their core coding abilities. In addition to pre-training, several approaches that fine-tune LLMs with instruction-following data (Ding et al., 2023) have also made substantial progress in improving models' understanding of user instructions and the quality of their responses. However, the effectiveness of post-training is heavily dependent on the availability of high-quality data (Xu et al., 2024a), and challenges of data collection and annotation remain difficult to overcome.

To address these challenges, some approaches propose various data flywheels to generate instruction data. Building on Self-Instruct, Chaudhary (2023) constructs Code Alpaca by prompting teacher LLMs to generate instructions in a few-shot setting. To further enhance the diversity and complexity of Code Alpaca, WizardCoder (Luo et al., 2024b) employs Evol-Instruct to evolve the original instructions. These methods apply general data augmentation to instruction construction, lacking specific design considerations for the code domain. Given that, recent methods specifically design frameworks for instruction generation tailored to code. For example, WaveCoder (Yu et al., 2024) collects raw code snippets and defines different tasks based on them. Similarly, InverseCoder (Wu et al., 2024b) prompts code LLMs to generate high-quality instructions based on the original code through techniques like code summarization and self-evaluation. These methods construct data for code in various ways, effectively enhancing the model's code generation capabilities. However, they still rely on existing datasets (Muennighoff et al., 2024) and calls for proprietary LLMs (e.g., GPT-3.5, GPT-4, etc.), making data collection costly. Additionally, the limited data sources and annotators constrain the diversity (Yu et al., 2023; Wang et al., 2023) of the data and inherit the system biases inherent in the limited pool of annotators (Wei et al., 2024).

The challenges mentioned above motivated us to propose **WarriorCoder**, which learns from expert battles to overcome current limitations. As illustrated in Figure 1, the attacker challenges the opponent within its area of expertise, and the target model learns from the winner of these pairwise competitions. Specifically, we design a completion-based method to mine the capabilities which the attacker has already mastered, then integrate Elo Rating and voting results to balance the local and global evaluation. This approach enables us to generate novel training data from scratch, incorporating the strengths of all the expert code LLMs, rather than relying on limited proprietary LLMs to expand existing datasets. Moreover, our method eliminates the need for human involvement and proprietary LLMs in the data collection, making it possible to collect high-quality, diverse data at a low cost. The main contributions of this paper are summarized as follows:

- We identify the limitations of current data flywheel and propose a new scalable paradigm where the target model learns from expert battles to solve them.
- We design a completion-based method for collecting instructions and introduce the Elo Rating system for evaluating responses, enabling the creation of high-quality and diverse training data at a low cost. Fine-tuned on this data, **WarriorCoder** incorporates the strengths of all the experts, achieving state-of-the-art performance compared to previous models of the same size, without relying on proprietary LLMs.
- Extensive experiments demonstrate the excellent performance of **WarriorCoder** on multiple code-related tasks, with ablation and analysis studies explaining how and why it works.

2 Related Work

2.1 Code LLMs

Code plays a crucial role in application areas for LLMs (Jain et al., 2024), attracting significant interest from both academia and industry. Codex (Chen et al., 2021), an LLM with 12 billion parameters, can solve 72.31% of complex Python programming problems. Following the success of Codex (Lyu et al., 2024), the rise of new code LLMs has demonstrated even greater capabilities, such as code generation and debugging, as model sizes continue to grow (Hou et al., 2023; Zan et al., 2023). Despite this impressive progress, the performance of current open-source models still lags behind that of proprietary ones (e.g., GPT-3.5, GPT-4, etc.), primarily because stronger models often keep their training data proprietary (Hui et al., 2024). As a result, the lack of publicly available code datasets remains a significant barrier to further development in this field.

2.2 Learning from Battles

Studying how people interact with LLMs in real-world scenarios is a pressing need for ensuring the alignment of LLMs (Chiang et al., 2024). The LMSYS Chatbot Arena (Zheng et al., 2024) has emerged as a groundbreaking initiative for exploring real-world LLM-user interactions, collecting and analyzing data from an open platform with over 240K votes. Experimental results demonstrate that the quality of data from the LMSYS Chatbot Arena is competitive with that of ShareGPT (Chiang et al., 2023), underscoring its value for training. Now more and more attentions are paid on learning from the battles between LLMs (Li et al., 2024; Myrzakhan et al., 2024; Bogomolov et al., 2024). However, collecting data through human online evaluations is both expensive and time-consuming. To address this, recent work has leveraged LLMs to provide their preferences when faced with different responses (Luo et al., 2024a; Zhao et al., 2024). Although these methods eliminate the need for human annotation during data collection, they still require pre-designed, high-quality instructions.

2.3 LLM as a Judge

Offering an automatic alternative to the scalability challenges inherent in human evaluation, the concept of LLM-as-a-judge has garnered significant public attention in recent years (Chiang and Lee, 2023). As large language models (LLMs) such as GPT-4 have demonstrated impressive capabilities, they are increasingly being considered for use in evaluating other machine-generated outputs (Weyssow et al., 2024). Experimental results by Thakur et al. (2024) show that these strong LLMs are capable of achieving a Cohen's Kappa coefficient over 80%, a metric typically used to assess the level of agreement between human raters. This performance level is comparable to the consensus found among human experts, highlighting the potential of LLMs to serve as reliable evaluators in various contexts. However, judge models often struggle with complex problems, and evaluating responses to such problems can be just as challenging as answering them (Wu et al., 2024a). Moreover, LLM-as-a-judge can introduce system biases, such as position bias, verbosity bias, and self-enhancement bias, which can undermine the fairness of the evaluation process (Chen et al., 2024a; Zheng et al., 2023a).

3 WarriorCoder: Learning from Expert Battles

In this section, we describe how **WarriorCoder** learns from expert battles. Unlike previous approaches that expand existing datasets by prompting a limited pool of proprietary LLMs, we construct an arena where state-of-the-art code LLMs compete against each other. Each model leverages its learned knowledge to challenge others, while judges evaluate the outcomes. The target model then learns from the winner of these pairwise competitions, progressively integrating the strengths of all competitors.

3.1 Competitors Setting

The capabilities of competitors determine the final performance of **WarriorCoder**. Theoretically, the more diverse and high-quality training data are derived from a larger and stronger pool of competitors. For this study, we select five leading open-source code experts from the BigCodeBench Leaderboard (Zhuo et al., 2024) - Athene-V2-Chat (Su et al., 2025), DeepSeek-Coder-V2-Lite-Instruct (DeepSeek-AI et al., 2024), Llama-3.3-70B-Instruct (Dubey et al., 2024), Qwen2.5-72B-Instruct (Hui et al., 2024), and QwQ-32B-Preview (Team, 2024). Notably, while **WarriorCoder** achieves state-of-the-art performance based solely on open-source code LLMs, it can also learn from powerful proprietary LLMs. In each round of the arena, only one pair of code experts is selected as competitors, while the remaining ones serve as judges.

3.2 Instruction Mining from Scratch

Considering a battle between LLM A and LLM B where A is the attacker and B is the defender. The first step of the arena is to use the strengths of A to challenge B, which makes it necessary to know what A has learned during its training process. However, almost all open-source LLMs keep their core data proprietary. Inspired by Magpie (Xu et al., 2024b), we design a completion-based method to mine the capabilities which the code LLMs have already mastered (① **Completion-based Instruction Mining**). Here we take Qwen2.5 (Hui et al., 2024) as an example. A conversation about writing Python code in the chat template of Qwen2.5 is:

```
<|im_start|>system
You are an AI assistant designed to provide helpful
on Python coding problems.<|im_end|>
<|im_start|>user
Write a Quicksort algorithm.<|im_end|>
<|im_start|>assistant
Here is the solution:
def quicksort(arr):
        ...
<|im_end|>
```

Such chat templates are pre-defined conversational structures to guide the interaction between the model and the user. Based on the LLMs' strong completion abilities, we feed only the prefix of the chat template into them, prompting the LLM to generate the user instructions:

```
<|im_start|>system
You are an AI assistant designed to provide re-
sponses on Python coding problems.<|im_end|>
<|im_start|>user
```

In this way, we can collect various instructions I the model has already learned under various generation settings (different values of temperature and top-p). Unlike traditional data synthesis, I is not synthesized by the models but directly sampled from their distributions, which avoids pattern overfitting and significant shifts in the output distribution (Chen et al., 2024b). However, these instructions may be repetitive, ambiguous, unclear, or too easy. To address these concerns, we deduplicate the data and adopt judges to assess their difficulty

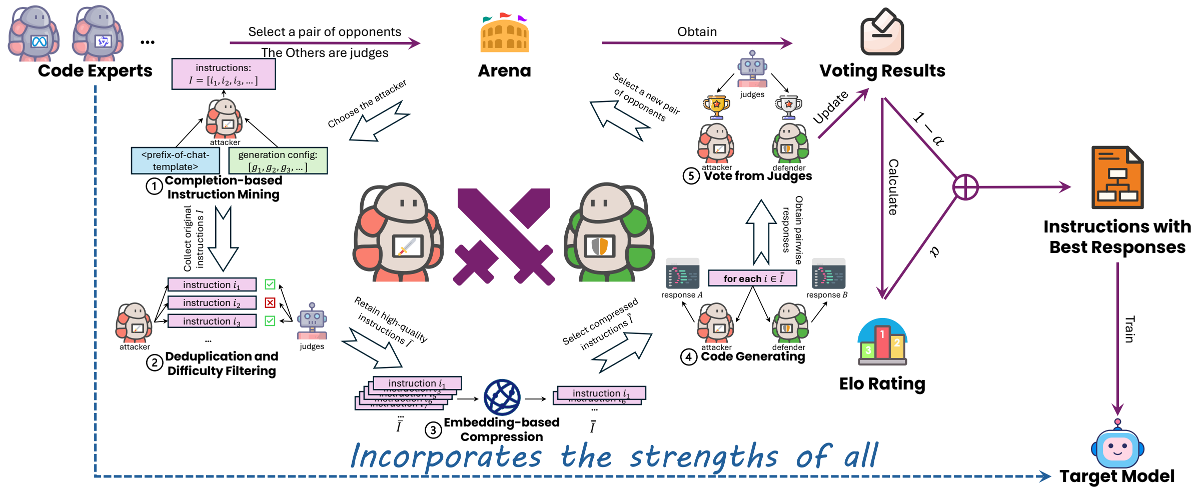
Code Experts
Select a pair of opponents
The Others are judges
Arena
Obtain
Voting Results
instructions: $I = [i_1, i_2, i_3, \ldots]$
attacker
<prefix-of-chat-template>
generation config: $[g_1, g_2, g_3, \ldots]$
① Completion-based Instruction Mining
Choose the attacker
Select a new pair of opponents
judges
attacker
defender
⑤ Vote from Judges
Update
$1-\alpha$
Calculate
α
Instructions with Best Responses
Collect original instructions I
instruction i_1
instruction i_2
instruction i_3
② Deduplication and Difficulty Filtering
Retain high-quality instructions $\bar{I}$
Select compressed instructions $\bar{\bar{I}}$
Obtain pairwise responses
for each $i \in \bar{\bar{I}}$
response A
response B
④ Code Generating
Elo Rating
Train
③ Embedding-based Compression
Incorporates the strengths of all
Target Model

Figure 2: The diagram of learning from expert battles. In each round of the arena, the attacker challenges the defender in its area of expertise under the evaluation of judges, and then the winner's response is added to the training data. In this manner, the target model gradually incorporates the strengths of all the code experts by fine-tuning on the data.

(② **Deduplication and Difficulty Filtering**). We divide the difficulty of instructions into four levels:

- **Excellent (9-10):** For instructions that are very clear, specific, and well-articulated. These instructions are particularly challenging and excellently designed to assess the AI's proficiency.
- **Good (6-8):** For instructions that are clear and specific instructions. These are not overly difficult to answer and moderately assess the AI's capabilities.
- **Average (3-5):** For instructions that are fairly clear and specific instructions. These instructions are easy to answer.
- **Poor (1-2):** For instructions that are ambiguous or unclear.

Only good and excellent instructions are considered during the following steps:

$$\bar{I} = \{i | i \in I \wedge d(i) \geq 6\} \tag{1}$$

where $d(i)$ is the difficulty of instruction i.

Then we compress the high-quality instructions $\bar{I}$ for the efficiency of post-training (③ **Embedding-based Compression**). To ensure the diversity and representativeness of instructions, we employ KCenterGreedy algorithm (Sener and Savarese, 2018) to select the final instructions $\bar{\bar{I}}$ based on the embedding model - *all-roberta-large-v1* (Liu et al., 2019).

3.3 Win-Loss Decision

The defender is required to respond to the attacker's question, while the attacker A must also provide an answer to its own instruction (④ **Code Generating**). Once both answers are collected, the judges (the rest LLMs in arena) will evaluate the correctness and helpfulness of the pairwise responses and vote for their preferred one (more details can be found in Appendix C). Then we can calculate the *local score* for each response:

$$x^i_{A>B} = \frac{t_A}{t_A + t_B} \quad x^i_{B>A} = \frac{t_B}{t_A + t_B} \tag{2}$$

where $x^i_{A>B}$ and $x^i_{B>A}$ are the local scores for A's and B's responses to the instruction i. $x^i_{A>B}$ represents the percentage of votes that candidate A receives, while $x^i_{B>A}$ similarly represents the percentage of votes that candidate B receives. t_A and t_B are the number of votes which A and B win.

However, relying solely on the *local score* to select the winner can be problematic. In some cases, a weaker model may receive more votes than a stronger one, even though its responses are not significantly better. This can occur because the *local score* may not fully capture the quality of the model's performance, especially in situations where the voting is influenced by factors, such as randomness or bias from LLM judges.

To address this limitation, we propose considering both local contingency and global consistency in the decision-making process. Instead of directly basing our analysis on the immediate voting outcomes, we introduce the concept of the *global score* — specifically, the Elo rating (Bai et al., 2022), which provides a more comprehensive reflection of a model's relative performance over time and across various evaluations. The Elo rating system, originally developed to calculate the relative skill levels of players in two-player games (such as chess), has been successfully adapted to assess the performance of competitors in a range of competitive scenarios, including esports and other skill-based games.

By incorporating the Elo rating, we account for both local performance in individual contests and global performance across multiple rounds, providing a more robust and accurate measure of a model's overall ability. This helps to mitigate the risk of weak models winning based on isolated, potentially unrepresentative votes:

$$\begin{aligned} X^{Elo}_{A>B} &= \frac{1}{1+10^{(R_B-R_A)/400}} \\ X^{Elo}_{B>A} &= \frac{1}{1+10^{(R_A-R_B)/400}} \end{aligned} \tag{3}$$

where $X^{Elo}_{A>B}$ and $X^{Elo}_{B>A}$ indicate the expected probabilities of A defeating B and B defeating A, respectively. R_A and R_B are the Elo rating of A and B, which are updated dynamically and iteratively. Given the battle result of A and B on an instruction i, we update them by:

$$\begin{aligned} R_A &\leftarrow R_A + K \times (s^i_{A>B} - X^{Elo}_{A>B}) \\ R_B &\leftarrow R_B + K \times (s^i_{B>A} - X^{Elo}_{B>A}) \end{aligned} \tag{4}$$

where $s^i_{A>B}$ and $s^i_{B>A}$ are the actual score of the battle result of player A and B (1 for a win, 0.5 for a draw, and 0 for a loss). The factor K controls the sensitivity of rating changes.

Based on Equation 2 and Equation 3, we can obtain the final score of A's response for instruction i:

$$e^i_A = \sum_{B \in Com \setminus A} \alpha X^{Elo}_{A>B} + (1-\alpha) x^i_{A>B} \tag{5}$$

where Com is the set of all the competitors and '$\setminus$' is the subtraction operation. α is the coefficient to balance the local contingency and global consistency.

3.4 Final Training

Each item in the constructed dataset consists of an instruction, responses from various strong LLMs, and their corresponding scores, which supports multiple post-training methods. We select the response with the highest score as the gold output to obtain **Instructions with Best Responses** in Figure 2 and use SFT to train the target model. In this manner, **WarriorCoder** integrates the strengths of all the code experts, as their expertise is embedded in the instructions and responses within the training data.

4 Experiments

4.1 Experimental Details

Backbones We use DeepSeekCoder-Base-6.7B (Guo et al., 2024) to initialize **WarriorCoder**. As for the competitors of expert battles, we choose strong open-source LLMs including Athene-V2-Chat (Su et al., 2025), DeepSeek-Coder-V2-Lite-Instruct (DeepSeek-AI et al., 2024), Llama-3.3-70B-Instruct (Dubey et al., 2024), Qwen2.5-72B-Instruct (Hui et al., 2024), and QwQ-32B-Preview (Team, 2024).

Datasets To evaluate the code generation capability of **WarriorCoder**, we conduct evaluations on HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), HumanEval+ (Liu et al., 2023), and MBPP+ (Liu et al., 2024). Besides, we also evaluate its code reasoning and libraries usage capabilities on CRUXEval (Gu et al., 2024) and DS-1000 (Lai et al., 2022). For a fair comparison, we use the same decoding strategies and generation configs as the previous work (Wei et al., 2024; Luo et al., 2024b; Yu et al., 2024).

Experimental Settings During the intruction minging, we adopt 9 different generation configs where temperature $t \in \{1.0, 1.1, 1.2\}$ and top-p $p \in \{0.99, 0.995, 1.0\}$. The final number of battle rounds is set to 70,000 and K is set to 40. α is set to 0.7 because we need the Elo Rating only when judges' opinions are divided on the evaluation. The detailed prompts can be found in Appendix B. As for the training stage, we conduct parallel training on 8 NVIDIA A800 80G GPUs. The global batch size is set to 512, and the number of total training steps is set to 448. We use a learning rate of 1×10^{-5} and a weight decay of 3×10^{-7}. Additionally, a WarmupLR scheduler with a warmup ratio of 0.2 is used.

Baselines The baselines consist of proprietary models, base models, and fine-tuned models.

Proprietary Models. These models, unlike open-source models, are developed, owned, and man-

	Models	HumanEval	HumanEval+	MBPP	MBPP+	Rely on proprietary LLMs?
Proprietary Models	Code-Davinci-002	47.0	-	58.1	-	-
	Code-Cushman-001	33.5	-	45.9	-	-
	GPT-3.5-Turbo	76.8	70.7	82.5	69.7	-
	GPT-4-Turbo	90.2	86.6	85.7	73.3	-
Base Models	DeepSeekCoder-Base (6.7B)	47.6	39.6	70.2	56.6	-
	CodeLlama (6.7B)	37.8	35.4	59.5	46.8	-
	StarCoder (15B)	34.1	29.3	55.1	46.1	-
Fine-tuned Models	CodeT5+ (16B)	31.7	26.2	54.6	44.4	✓
	WizardCoder-CL (6.7B)	48.2	40.9	56.6	47.1	✓
	WizardCoder-SC (15B)	51.9	45.1	61.9	50.6	✓
	Magicoder-DS (6.7B)	66.5	60.4	75.4	61.9	✓
	MagicoderS-DS (6.7B)	76.8	70.7	75.7	64.4	✓
	Magicoder-CL (6.7B)	60.4	55.5	64.2	52.6	✓
	MagicoderS-CL (6.7B)	70.7	66.5	68.4	56.6	✓
	WaveCoder-DS (6.7B)	72.0	-	63.6	-	✓
	WaveCoder-CL (6.7B)	48.1	-	47.2	-	✓
	WaveCoder-SC (15B)	50.5	-	51.0	-	✓
Ours	WarriorCoder (6.7B)	**80.5** (+32.9)	**75.6** (+36.0)	**76.2** (+6.0)	**64.8** (+8.2)	✗

Table 1: The pass@1(%) results on the code generation benchmarks (Humaneval, Humaneval+, MBPP and MBPP+).

aged by a private entity or organization. They are trained on specialized or private datasets that are not publicly available to serve specific business needs or objectives. Access to these models is usually based on API calls. Proprietary Models include Code-Davinci-002, Code-Cushman-001, GPT-3.5-Turbo (Ouyang et al., 2022) and GPT-4-Turbo (OpenAI et al., 2024).

Base Models. They are the foundational, pre-trained models that serve as the core for further fine-tuning or adaptation to code tasks. Base Models include DeepSeekCoder-Base (Guo et al., 2024), CodeLlama (Rozière et al., 2023), and StarCoder (Li et al., 2023).

Fine-tuned Models. These models are initially pre-trained on a large, general-purpose dataset and then fine-tuned on a smaller, code-specific dataset. This two-step process enhances the model's performance on coding tasks by enabling it to leverage both broad general knowledge and more focused, domain-specific expertise. Fine-tuned models include CodeT5+, DeepSeek-Coder-Instruct (Guo et al., 2024), WizardCoder (Luo et al., 2024b), Magicoder (Wei et al., 2024), and WaveCoder (Yu et al., 2024). The suffixes -DS, -CL, and -SC denote the base models DeepSeekCoder-Base, CodeLlama-Python, and StarCoder, respectively.

4.2 Main Results

The results on the code generation benchmarks are summarized in Table 1. **WarriorCoder** achieves SOTA performance, with a pass@1 accuracy of 80.5% (75.6%) in HumanEval (HumanEval +) and 76.2% (64.8%) in MBPP (MBPP +), surpassing all other fine-tuned models. Particularly, it shows a significant boost over on HumanEval and HumanEval+ (with gains of 32.9 and 36.0, respectively). **WarriorCoder** also outperforms MagicoderS-DS, Magicoder-DS and WaveCoder-DS, which share the same backbone architecture and similar amounts of training data. This substantial performance gap highlights the effectiveness of our approach in generating higher-quality training data, providing a clear advantage over models that rely on similar foundational setups.

Moreover, **WarriorCoder** also achieves excellent performances on the code reasoning benchmark and libraries usage benchmark. As shown in Table 2, **WarriorCoder** outperforms a range of open-source models, including those with sizes up to 34B, in pass@1 accuracy and achieves better pass@5 accuracy compared to GPT-3.5-Turbo (66.5% vs 63.2% on CRUXEval-I and 66.3% vs 59.3% on CRUXEval-O). Table 3 shows that **WarriorCoder** outperforms all baselines on most of the libraries, especially on SciPy, Sklearn and Tensorflow (33.0% , 39.1% , and 42.2%, respectively). These results highlight **WarriorCoder** as a powerful paradigm - a data flywheel that absorbs expertise from multiple code domains. Our approach significantly enhances the target model's ability to generalize across various tasks, demonstrating its superiority in leveraging diverse data sources to drive performance improvements.

Addiontionally, previous data flywheels typically rely on augmentations and annotations generated using proprietary LLMs and specially designed prompts. In contrast, our approach does not require pre-existing datasets, diverse handwritten prompts or proprietary LLMs. Despite this, we get competitive results that rival those of advanced

Models		CRUXEval-I		CRUXEval-O	
		Pass@1	Pass@5	Pass@1	Pass@5
Proprietary Models	GPT-4-Turbo	69.8	76.8	68.7	73.0
	GPT-3.5-Turbo	49.0	63.2	49.4	59.3
	Claude-3-Opus	64.2	-	65.8	-
Open-source Models	StarCoder (6.7B)	29.7	47.3	32.2	44.9
	StarCoder (15B)	31.3	49.2	34.2	47.1
	DeepSeekCoder-Instruct (6.7B)	37.4	53.3	41.2	52.8
	CodeLlama-Python (6.7B)	37.3	57.0	35.9	48.8
	CodeLlama-Python (13B)	39.7	56.9	39.8	52.5
	CodeLlama-Python (34B)	**43.9**	59.5	41.4	52.9
	Mistral (6.7B)	35.0	52.3	34.3	48.6
	WizardCoder (13B)	36.5	51.6	41.3	52.4
	WizardCoder (34B)	42.7	57.5	43.4	53.8
	Magicoder(6.7B)	41.7	62.4	44.4	57.5
Ours	WarriorCoder (6.7B)	42.9	**66.5**	**45.4**	**66.3**

Table 2: The pass@1(%) and pass@5(%) results on the code reasoning benchmark (CRUXEval).

Models	Matplotlib (155)	NumPy (220)	Pandas (291)	PyTorch (68)	SciPy (106)	Sklearn (115)	TensorFlow (45)	Overall (1000)
INCODER (6.7B)	28.3	4.4	3.1	4.4	2.8	2.8	3.8	7.4
CodeGen-Mono (16B)	31.7	10.9	3.4	7.0	9.0	10.8	15.2	11.7
Code-Cushman-001	40.7	21.8	7.9	12.4	11.3	18.0	12.2	18.1
StarCoder (15B)	51.7	29.7	11.4	21.4	20.2	29.5	24.5	26.0
WizardCoder-SC (15B)	55.2	33.6	16.7	26.2	24.2	24.9	26.7	29.2
CodeLlama-Python (6.7B)	55.3	34.5	16.4	19.9	22.3	17.6	28.5	28.0
WizardCoder-CL (6.7B)	53.5	34.4	15.2	25.7	21.0	24.5	28.9	28.4
Magicoder-CL (6.7B)	54.6	34.8	19.0	24.7	25.0	22.6	28.9	29.9
MagicoderS-CL (6.7B)	**55.9**	40.6	**28.4**	40.4	28.8	35.8	37.6	37.5
WarriorCoder (6.7B)	55.5	**41.8**	26.1	**41.2**	**33.0**	**39.1**	**42.2**	**38.1**

Table 3: The pass@1(%) results on the benchmark for using Python libraries in data science (DS-1000).

Task	Percentage(%)	Definition
Code Generation	51.4	Generating source code based on certain specifications or requirements.
Code Debugging	12.2	Identifying, diagnosing, and fixing errors or bugs in a code snippet.
Code Optimization	3.8	Improving a program's performance, efficiency, or resource usage without changing its functionality.
Code Reasoning	2.9	Predicting the output based on the given input or predicting the input from the known output.
Code Analysis	6.6	Analyzing, understanding, and explaining how a piece of code works.
Theoretical Explanation	22.2	Answering the questions about principles, theories, and properties of programming language.
Code Transpile	0.9	Converting source code from one programming language into another programming language.

Table 4: The proportion of different tasks in the training data.

#Num	HumanEval	HumanEval+	MBPP	MBPP+
1	75.4	72.6	73.3	62.4
2	77.2	73.3	74.5	62.9
5	80.5	75.6	76.2	64.8

Table 5: The results observed when learning from varying numbers of experts.

proprietary code experts. This highlights the effectiveness of our data flywheel, demonstrating the feasibility of collecting high-quality data at a low cost.

4.3 Ablation Study

Table 5 presents the results observed when the target model learns from varying numbers of experts. The target model shows a significant improvement when learning from just one code LLM, indicating that even a single code expert enables it to acquire a specific set of knowledge. However, as the number of experts increases, **WarriorCoder** benefits from learning across all expert code LLMs. As a result, the model trained with 5 code LLMs outperforms others across all four benchmarks, demonstrating the advantages of integrating knowledge from multiple specialized experts.

4.4 Data Analysis

4.4.1 Dependence Analysis

Figure 3 illustrates the overlap between the instructions mined from expert LLMs and those from widely used code training datasets, measured using the ROUGE score. The majority of the mined instructions have a ROUGE score of less than 0.3, suggesting they are largely distinct from those in existing datasets. Notably, no mined instructions exceed a ROUGE score of 0.6, further emphasiz-

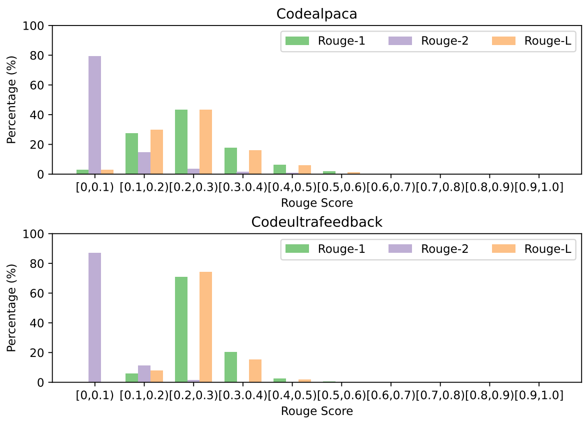

Figure 3: The overlapping rate between the mined instructions and existing training datasets.

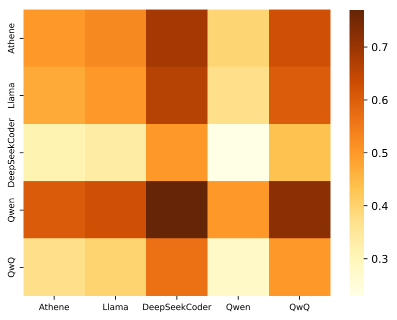

Figure 4: The heatmap of win rates of the selected code experts.

ing that the mined instructions are drawn from the internal distribution of expert LLMs, rather than being simple replications or extensions of the training data. Consequently, these instructions exhibit a higher degree of independence, making them particularly valuable for training, as they provide novel examples that can enhance the target model's capabilities.

4.4.2 Diversity Analysis

Table 4 reveals the distribution of different tasks in the training data. The range of instructions covers a variety of tasks, ensuring that **WarriorCoder** can generalize effectively across multiple benchmarks. Notably, while Code Reasoning represents only 2.9% of the entire dataset, **WarriorCoder** still achieves outstanding performance on CRUXEval, highlighting the potential of the framework that learns from expert battles. Furthermore, Figure 4 illustrates the battle results between the five selected code experts. Even though an expert may have the highest Elo Rating, it is not necessarily the best performer on all tasks. However, **WarriorCoder** learns from the winner of each instruction, thereby diversifying the target responses.

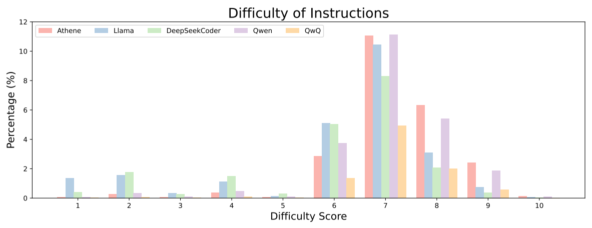

Figure 5: The proportion of difficulties of mined instructions. As mentioned in Section 3.2, the difficulties of instructions are divided into four levels: excellent (9-10), good(6-8), average(3-5) and poor(1-2).

4.4.3 Difficulty Analysis

Figure 5 shows the difficulty distribution of the mined instructions, offering insights into the internal knowledge of the code experts. Most instructions fall within the 'good' level, with scores between 6 and 8. Instructions rated as 'excellent' (scores 9-10) constitute only a small portion of the dataset, indicating that highly complex or advanced tasks are relatively rare. Instructions with scores below 6 are excluded from the training set, as they tend to be either too easy or overly ambiguous. Such instructions are considered detrimental to the training stage, as they may not provide meaningful learning signals and could undermine the model's performance and generalization ability. More examples are listed in Appendix A.

5 Conclusion

This paper highlights the limitations of existing data flywheels for code LLMs that primarily rely on pre-existing datasets and annotations from a limited pool of proprietary LLMs, leading to a lack of data diversity and reinforces the systemic biases. Even more concerning is the fact that many current open-source expert code LLMs keep their training data proprietary, further restricting access to diverse and high-quality data sources. To address these challenges, we propose WarriorCoder, which learns from expert battles, enabling the absorption of each expert's strengths. Unlike existing methods that expand and refine datasets, we construct training data from scratch and achieve SOTA performances on multiple benchmarks without the need for pre-existing datasets and costly annotations. Furthermore, our approach can potentially be applied to other complex tasks besides coding in the future.

Limitations

In this paper, we propose a novel training paradigm in which the target model learns from expert battles, aiming to overcome the limitations of current data flywheels. While we can generate high-quality and diverse data from scratch at a low cost, the battle process can become time-consuming when the number of experts is large. Exploring more efficient and effective competition modes is a promising direction for future work.

References

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, Nicholas Joseph, Saurav Kadavath, Jackson Kernion, Tom Conerly, Sheer El Showk, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Tristan Hume, Scott Johnston, Shauna Kravec, Liane Lovitt, Neel Nanda, Catherine Olsson, Dario Amodei, Tom B. Brown, Jack Clark, Sam McCandlish, Chris Olah, Benjamin Mann, and Jared Kaplan. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback. *CoRR*, abs/2204.05862.

Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria Tigina, Yaroslav Golubev, Alexander Kovrigin, Arie van Deursen, Maliheh Izadi, and Timofey Bryksin. 2024. Long code arena: a set of benchmarks for long-context code models. *CoRR*, abs/2406.11612.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. `https://github.com/sahil280114/codealpaca`.

Guiming Chen, Shunian Chen, Ziche Liu, Feng Jiang, and Benyou Wang. 2024a. Humans or llms as the judge? A study on judgement bias. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 8301–8327. Association for Computational Linguistics.

Jie Chen, Yupeng Zhang, Bingning Wang, Xin Zhao, Ji-Rong Wen, and Weipeng Chen. 2024b. Unveiling the flaws: Exploring imperfections in synthetic data and mitigation strategies for large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*, pages 14855–14865. Association for Computational Linguistics.

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

David Cheng-Han Chiang and Hung-yi Lee. 2023. Can large language models be an alternative to human evaluations? In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 15607–15631. Association for Computational Linguistics.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality.

Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Banghua Zhu, Hao Zhang, Michael I. Jordan, Joseph E. Gonzalez, and Ion Stoica. 2024. Chatbot arena: An open platform for evaluating llms by human preference. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *CoRR*, abs/2406.11931.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. Enhancing chat language models by scaling high-quality instructional conversations. In *Proceedings of the 2023 Conference on Empirical*

Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023, pages 3029–3051. Association for Computational Linguistics.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. 2024. The llama 3 herd of models. *CoRR*, abs/2407.21783.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John C. Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *CoRR*, abs/2308.10620.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

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

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason T. Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *Trans. Mach. Learn. Res.*, 2023.

Tianle Li, Wei-Lin Chiang, Evan Frick, Lisa Dunlap, Tianhao Wu, Banghua Zhu, Joseph E. Gonzalez, and Ion Stoica. 2024. From crowdsourced data to high-quality benchmarks: Arena-hard and benchbuilder pipeline. *CoRR*, abs/2406.11939.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chat-

GPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. 2024. Evaluating language models for efficient code generation. In *First Conference on Language Modeling*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Qingwei Lin, Jianguang Lou, Shifeng Chen, Yansong Tang, and Weizhu Chen. 2024a. Arena learning: Build data flywheel for llms post-training via simulated chatbot arena. *CoRR*, abs/2407.10627.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024b. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Michael R. Lyu, Baishakhi Ray, Abhik Roychoudhury, Shin Hwei Tan, and Patanamon Thongtanunam. 2024. Automatic programming: Large language models and beyond. *CoRR*, abs/2405.02213.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Aidar Myrzakhan, Sondos Mahmoud Bsharat, and Zhiqiang Shen. 2024. Open-llm-leaderboard: From multi-choice to open-style questions for llms evaluation, benchmark, and arena. *CoRR*, abs/2406.07545.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Ozan Sener and Silvio Savarese. 2018. Active learning for convolutional neural networks: A core-set approach. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

DiJia Su, Hanlin Zhu, Yingchen Xu, Jiantao Jiao, Yuandong Tian, and Qinqing Zheng. 2025. Token assorted: Mixing latent and text tokens for improved language model reasoning. *Preprint*, arXiv:2502.03275.

Qwen Team. 2024. Qwq: Reflect deeply on the boundaries of the unknown.

Aman Singh Thakur, Kartik Choudhary, Venkat Srinik Ramayapally, Sankaran Vaidyanathan, and Dieuwke Hupkes. 2024. Judging the judges: Evaluating alignment and vulnerabilities in llms-as-judges. *CoRR*, abs/2406.12624.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 13484–13508. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Martin Weyssow, Aton Kamanda, and Houari A. Sahraoui. 2024. Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences. *CoRR*, abs/2403.09032.

Tianhao Wu, Weizhe Yuan, Olga Golovneva, Jing Xu, Yuandong Tian, Jiantao Jiao, Jason Weston, and Sainbayar Sukhbaatar. 2024a. Meta-rewarding language models: Self-improving alignment with llm-as-a-meta-judge. *CoRR*, abs/2407.19594.

Yutong Wu, Di Huang, Wenxuan Shi, Wei Wang, Lingzhe Gao, Shihao Liu, Ziyuan Nan, Kaizhao Yuan, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, Yewen Pu, Dawei Yin, Xing Hu, and Yunji Chen. 2024b. Inversecoder: Unleashing the power of instruction-tuned code llms with inverse-instruct. *CoRR*, abs/2407.05700.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024a. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Zhangchen Xu, Fengqing Jiang, Luyao Niu, Yuntian Deng, Radha Poovendran, Yejin Choi, and Bill Yuchen Lin. 2024b. Magpie: Alignment data synthesis from scratch by prompting aligned llms with nothing. *CoRR*, abs/2406.08464.

Yue Yu, Yuchen Zhuang, Jieyu Zhang, Yu Meng, Alexander J. Ratner, Ranjay Krishna, Jiaming Shen, and Chao Zhang. 2023. Large language model as attributed training data generator: A tale of diversity and bias. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024. Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 5140–5153. Association for Computational Linguistics.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2023. Large language models meet nl2code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 7443–7464. Association for Computational Linguistics.

Ruochen Zhao, Wenxuan Zhang, Yew Ken Chia, Deli Zhao, and Lidong Bing. 2024. Auto arena of llms: Automating LLM evaluations with agent peer-battles and committee discussions. *CoRR*, abs/2405.20267.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2024. Lmsys-chat-1m: A large-scale real-world LLM conversation dataset. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023a. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023b. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *CoRR*, abs/2303.17568.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro von Werra. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *CoRR*, abs/2406.15877.

You are an AI assistant designed to provide helpful, step-by-step guidance on coding problems. The user will ask you a wide range of Python coding questions. Your purpose is to assist users in understanding coding concepts, working through code, and arriving at the correct solutions.

Table 6: The prompt for instruction mining.

A Case Study

Here we show two examples of mined instructions. The first has a score of 6:

> I'm trying to combine two dictionaries into one and eliminate duplicate values for a given key the two dictionaries may have different structures, eg.
> dictA = {'cat1':{'cat2':'A'}, 'cat2':{'cat3':'B'}...}
> dictB = {'cat1':{'cat2':'C'}}
> Combining with a recursive function seems the best option but I can't seem to get it right.
> **Please Help.**

And the second has a score of 9:

> **Function to Get the Index of an Element in a 2D List (List of Lists)**
> ====================================
> Create a function named 'get_index_2d' that accepts three parameters:
> - a 2D list 'matrix'
> - an element to search for 'target'
> - a default value to return 'default'
>
> If the 'target' exists in the 'matrix', the function should return its index (a tuple containing row, column).
> If the 'target' is not found in the 'matrix', the function should return the 'default' value.
> """python
> def get_index_2d(matrix, target, default=None):
>   # implement function
> """
> ### Example Use Cases:

Matrix	Target	Default	Expected
[[1,2],[3,4]]	4	None	(1,1)
[[1,2],[3,4]]	5	None	None
[[1,2],[3,4]]	1	(0,-1)	(0,0)
[['a','b'], ['c','d']]	'd'	(-1,-1)	(1,1)

Both instructions are clear and specific, providing a solid foundation for tackling the problem. However, the first instruction is more straightforward, with fewer requirements and a greater emphasis on practical tips and strategies to handle the

This is a chatbot arena. You will be given assistant A's answer, and assistant B's answer. Please act as an impartial judge and evaluate the capability of two AI assistants. You should choose the assistant that follows instructions and answers questions better. Your evaluation should consider factors such as helpfulness, relevance, and accuracy. Begin your evaluation by comparing the responses of the two assistants and provide a short explanation. 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, choose the one that is straight-to-the-point instead of unnecessarily verbose. When the two candidates perform equally well, choose the SHORTER answer. Do not favor certain names of the assistants. Be as objective as possible. After providing your explanation concisely within 200 words, output your final verdict by strictly following this format: "[[A]]" if assistant A is better, "[[B]]" if assistant B is better, and "[[Tie]]" for a tie. Finish your judgement within 300 words. [[User Question]] {instruction} [[The Start of Assistant A's Answer]] {#A's Answer} [[The End of Assistant A's Answer]] [[The Start of Assistant B's Answer]] {#B's Answer} [[The End of Assistant B's Answer]]

Table 7: The prompt for the evaluation of pairwise competitions.

task. In contrast, the second instruction introduces more complex conditions and does not include any guidance, making the problem more challenging to solve.

B Prompts for Instruction Mining and Evaluation

Table 6 and Table 7 show the prompts for instruction mining and evaluation. To maintain impartiality in the evaluation process, we withhold the names of the opponents from the judges, thus avoiding potential system biases. Additional rules will be discussed in the next section.

C Rules for the Fairness of Evaluation

To ensure impartiality in the evaluation process, we establish a set of rules, including **order shuffling**, **suspicion averting**, and **offensive-defense balance**.

Order shuffling refers to the practice of randomizing the order in which responses from the **attacker (A)** and **defender (B)** appear in the evaluation prompt. This helps mitigate any positional bias that may arise if a particular position is favored by certain language models.

Suspicion averting ensures that competitors do not evaluate their own responses, preventing any potential bias in favor of their own generated answers.

Offensive-defense balance guarantees that all competitors have an equal number of offensive and defensive turns, maintaining fairness in the evaluation process.