SelfCodeAlign: Self-Alignment for Code Generation

Yuxiang Wei[1] **Federico Cassano**[2,7] **Jiawei Liu**[1] **Yifeng Ding**[1] **Naman Jain**[3] **Zachary Mueller**[5]
Harm de Vries[4] **Leandro von Werra**[5] **Arjun Guha**[2,6] **Lingming Zhang**[1]

[1]University of Illinois Urbana-Champaign [2]Northeastern University
[3]University of California, Berkeley [4]ServiceNow Research [5]Hugging Face [6]Roblox [7]Cursor AI

`{ywei40,lingming}@illinois.edu` `{cassano.f,a.guha}@northeastern.edu`
https://github.com/bigcode-project/selfcodealign

Abstract

Instruction tuning is a supervised fine-tuning approach that significantly improves the ability of large language models (LLMs) to follow human instructions. For programming tasks, most models are finetuned with costly human-annotated instruction-response pairs or those generated by large, proprietary LLMs, which may not be permitted. We propose SelfCodeAlign, the first fully transparent and permissive pipeline for self-aligning code LLMs without extensive human annotations or distillation. SelfCodeAlign employs the same base model for inference throughout the data generation process. It first extracts diverse coding concepts from high-quality seed snippets to generate new tasks. It then samples multiple responses per task, pairs each with test cases, and validates them in a sandbox environment. Finally, passing examples are selected for instruction tuning. In our primary experiments, we use SelfCodeAlign with CodeQwen1.5-7B to generate a dataset of 74k instruction-response pairs. Finetuning on this dataset leads to a model that achieves a 67.1 pass@1 on HumanEval+, surpassing CodeLlama-70B-Instruct despite being ten times smaller. Across all benchmarks, this finetuned model consistently outperforms the original version trained with OctoPack, the previous state-of-the-art method for instruction tuning without human annotations or distillation. Additionally, we show that SelfCodeAlign is effective across LLMs of various sizes, from 3B to 33B, and that the base models can benefit more from alignment with their own data distribution. We further validate each component's effectiveness in our pipeline, showing that SelfCodeAlign outperforms both direct distillation from GPT-4o and leading GPT-3.5-based distillation methods, such as OSS-Instruct and Evol-Instruct. SelfCodeAlign has also led to the creation of StarCoder2-Instruct, the first fully transparent, permissively licensed, and self-aligned code LLM that achieves state-of-the-art coding performance. Overall, SelfCodeAlign shows for the first time that a strong instruction-tuned code LLM can result from self-alignment rather than distillation.

1 Introduction

Recent studies have demonstrated the outstanding performance of large language models (LLMs) [33, 40, 19, 57, 45, 69, 8, 70] in various code-related tasks, *e.g.,* program synthesis [8, 3], program repair [78, 27, 24, 79, 73], code optimization [59, 9], code completion [11, 40, 19], code translation [56, 1, 51], software testing [32, 10, 42, 77], and software agents [80, 67, 75, 37]. The reason is that modern LLMs are pre-trained over trillions of code tokens in the wild using various training objectives (as such next-token prediction [52]), making the base models natively good at understanding and generating code snippets. Furthermore, to fully unleash the power of LLMs, the base models are typically further fine-tuned on high-quality instruction-following data to boost their performance in following natural language instructions and solving more general software engineering tasks [25]. This step is known as *instruction tuning* [50].

Curating high-quality data for instruction tuning is crucial yet challenging. One source of acquiring instruction data is to employ human annotation [50]. For example, Llama-3 [14] uses a corpus of 10 million human-annotated examples in instruction tuning. Due to the high cost of human annotation, knowledge distillation is widely adopted to train a weaker LLM with outputs generated by stronger LLMs [18]. However, distillation may violate the terms of service [48, 17, 2] of proprietary LLMs and the prerequisite of using a stronger LLM limits its generalizability. Therefore, recent proposals focus on instruction tuning without relying on human annotation or distillation [34, 60, 82]. One cornerstone work along this direction is SELF-INSTRUCT [68], which finetunes GPT-3 with self-generated instruction data using in-context learning.

There is a growing number of instruction-tuned open-source LLMs in the code domain. However, some models, such as DeepSeek-Coder [19], Llama-3 [14], and CodeQwen1.5 [64], either use proprietary data or do not disclose their instruction-tuning strategies. Others, including WizardCoder [41], Magicoder [72], WaveCoder [81], and OpenCodeInterpreter [83], rely on knowledge distillation. The only exception is OctoCoder [43], which is instruction-tuned over heavily filtered GitHub commits, with commit messages as instructions and the changed code as responses, as well as data from OpenAssistant, a human-generated corpus of user-assistant conversations [29]. Despite its transparency and permissive licensing, OctoCoder's performance, at 32.9 HumanEval+ pass@1, lags behind other mainstream code LLMs. Meanwhile, previous attempts at applying SELF-INSTRUCT for code generation have resulted in performance degradation over training on natural instruction-response pairs [43]. Our findings imply that effective self-alignment requires a combination of data diversity control and response validation, which is not present in the traditional SELF-INSTRUCT approach.

In this paper, we propose SelfCodeAlign, the first fully transparent pipeline to successfully self-align base code LLMs with purely self-generated instruction data. First, SelfCodeAlign extracts diverse coding concepts from high-quality seed functions in The Stack V1 [28], a large corpus of permissively licensed code. Next, using these concepts, we prompt the base model to generate new coding tasks through in-context learning. We then instruct the base model to produce multiple responses for each task, each paired with test cases for self-validation. Finally, we select only the instruction-response pairs that pass the test cases. This method ensures the model practices various coding concepts and validates the consistency between instructions and responses.

To evaluate our method, we train CodeQwen1.5-7B, a state-of-the-art open-source base LLM for code, on both a dataset generated with SelfCodeAlign and OctoPack, a naturally-generated and meticulously-filtered dataset used for training OctoCoder [43]. We benchmark both, along with OctoCoder and other models, on a series of tasks: code generation (both function- and class-level) [38, 21, 76, 13], data science programming [30], and code editing [6]. On all tasks, training CodeQwen with SelfCodeAlign significantly improves performance over the base model and over training it on OctoPack. For instance, on HumanEval+, our model achieves a pass@1 score of 67.1, 21.4 points higher than CodeQwen1.5-7B and 16.5 points higher than CodeQwen1.5-7B-OctoPack. This highlights the effectiveness of our synthetic data generation method compared to natural data in enhancing the capabilities of code LLMs.

In the component analysis, we justify the different components of the pipeline. We demonstrate that SelfCodeAlign is general to different LLMs whose sizes range from 3B to 33B. In particular, we find that a base LLM could learn more effectively from data within its own distribution than a shifted distribution from a teacher LLM. Additionally, we show that seed selection, concept generation, and execution filtering all contribute positively to the pipeline. Furthermore, on HumanEval+, SelfCodeAlign (67.1 pass@1) outperforms state-of-the-art, GPT-3.5-Turbo-based distillation methods, including OSS-Instruct [72] (61.6) and Evol-Instruct [65] (59.1), as well as direct output distillation from GPT-4o [49] (65.9).

SelfCodeAlign has also led to the creation of StarCoder2-Instruct, the first fully transparent, permissively licensed, and self-aligned code LLM that achieves state-of-the-art coding performance. We discuss StarCoder2-Instruct in Appendix A.

Overall, we make the following main contributions: *(i)* We introduce SelfCodeAlign, the first fully transparent and permissive pipeline for self-aligning code LLMs to follow instructions. Our method does not rely on extensive human annotations or distillation from larger models. *(ii)* We generate a series of datasets using SelfCodeAlign and train multiple models on these datasets, which will all be released to the public. *(iii)* We thoroughly evaluate our method on a multitude of tasks, showing strong performance across all the evaluated models. *(iv)* Our experiments demonstrate that training models on their own data can be more effective than using data from stronger, but distributionally different, teacher models when they don't have a huge performance gap. *(v)* Finally, we run a comprehensive component analysis that verifies the positive contribution of each component in SelfCodeAlign.

2 SelfCodeAlign: Self-Alignment for Code Generation

Figure 1 illustrates an overview of our SelfCodeAlign technique. It first generates diverse instructions by extracting coding concepts from high-quality seed snippets. This process resembles OSS-Instruct [72], which employs GPT-3.5-Turbo to convert random snippets into instructions. However, our method uses the base model exclusively and incorporates a separate concept generation phase that we prove beneficial in §4.3. SelfCodeAlign then generates several responses for each task, pairing each with test cases for sandbox execution, and finally chooses passing examples for instruction tuning. Example outputs from each step are listed in Appendix D.1. In the following sections, we provide detailed explanations of these steps.

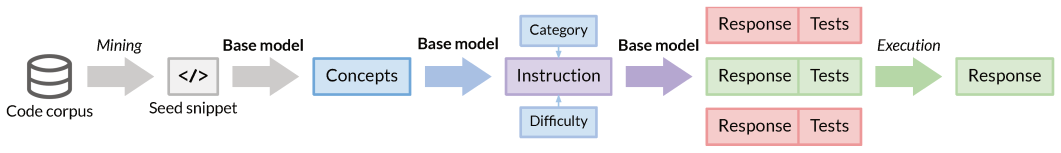

Figure 1: Overview of SelfCodeAlign.

2.1 Seed Snippets Collection

SelfCodeAlign starts by collecting a set of seed code snippets from The Stack V1. In this step, it's crucial to ensure that the seed snippets are diverse and high-quality, as they will be used as the starting point for generating instructions and responses. To collect the seed snippets, we extract all Python functions with docstrings from The Stack V1, and then apply a series of filtering rules to ensure the quality of the seed snippets. In total, we collect 250k Python functions from 5M functions with docstrings in The Stack V1, which were filtered by running the Pyright type checker, removing benchmark items, filtering out functions with poor documentation, and removing near-duplicates. Appendix B details this process in depth.

2.2 Diverse Instruction Generation

After collecting the seed functions, we perform Self-OSS-Instruct, our adaptation of OSS-Instruct [72] for self-alignment, to generate diverse instructions. In detail, we employ in-context learning to let the base model self-generate instructions from the given seed code snippets. This process utilizes 21 carefully designed few-shot examples listed in Appendix E. The instruction generation procedure is divided into the following two steps:

- **Concepts extraction:** For each seed function, we prompt the base model to produce a list of code concepts present within the function. Code concepts refer to the foundational principles and techniques used in programming, such as pattern matching and data type conversion.
- **Instruction generation:** We then prompt the base model to self-generate a coding task conditioned on the identified code concepts and two additional attributes, difficulty (easy/medium/hard) and category (function/class/program implementation), which we randomly sample to enrich the diversity of the generated instructions.

2.3 Response Generation and Self-Validation

Given the instructions generated from Self-OSS-Instruct, our next step is to match each instruction with a high-quality response. Prior practices commonly rely on distilling responses from stronger teacher models, such as GPT-4, which hopefully exhibit higher quality. However, distilling proprietary models leads to non-permissive licensing and a stronger teacher model might not always be available. More importantly, teacher models can be wrong as well, and the distribution gap between teacher and student can be detrimental.

We propose to self-align the base model by explicitly instructing the model to generate tests for self-validation after it produces a response interleaved with natural language. This process is similar to how developers test their code implementations. Specifically, for each instruction, the base model samples multiple outputs of the format *(response, tests*), and we filter out those responses falsified by the test execution under a sandbox environment. We then randomly select one passing response per instruction to the final instruction tuning dataset.

3 Main Evaluation

In this section, we comprehensively evaluate SelfCodeAlign over a diverse set of coding tasks:

- **Function generation (§3.1):** Given a natural-language description, LLMs are asked to generate a self-contained function whose correctness and efficiency is checked through test execution [8, 3, 38, 21, 76, 39].
- **Class generation (§3.2):** Given a code skeleton with both class- and method-level information, LLMs are asked to generate the class and its methods [13].
- **Data science programming (§3.3):** Given a description of a data science task and a partial code snippet, LLMs are asked to complete the code snippet to pass corresponding tests [30].
- **File-level code editing (§3.4):** Provided with the contents of a file, the model is asked to edit the program following a natural language instruction [6].

3.1 Function-level Code Generation

Table 1: Pass@1 (%) of different LLMs on EvalPlus computed using greedy decoding.

Model	Instruction data	Benchmark		Artifact		
		HumanEval+	MBPP+	Transparent	Non-proprietary	Non-distilled
GPT-4-Turbo [47]	Not disclosed	81.7	70.7	○	○	●
Mistral Large [22]	Not disclosed	62.8	56.6	○	○	●
Gemini Pro [63]	Proprietary	55.5	57.9	○	○	●
Llama3-70B-Instruct [14]	Proprietary	70.7	66.4	○	○	●
CodeLlama-70B-Instruct [57]	Proprietary	65.2	61.7	○	○	●
WizardCoder-33B-v1.1 [41]	GPT distillation	73.2	66.9	○	●	○
OpenCodeInterpreter-DS-33B [83]	GPT distillation	73.8	67.7	●	●	○
Magicoder-S-DS-6.7B [72]	GPT distillation	70.7	65.4	●	●	○
DeepSeekCoder-33B-Instruct [19]	Not disclosed	75.0	66.7	○	-	-
CodeQwen1.5-7B-Chat [64]	Not disclosed	77.7	67.2	○	-	-
Snowflake Arctic (480B) [55]	Not disclosed	64.3	64.3	○	-	-
Mixtral-8x22B-Instruct-v0.1 [23]	Not disclosed	70.1	62.9	○	-	-
Command-R+ (104B) [16]	Not disclosed	56.7	58.6	○	-	-
Mixtral-8x7B-Instruct-v0.1 [23]	Not disclosed	39.6	49.7	○	-	-
OctoCoder-16B [43]	Publicly available	32.9	49.1	●	●	●
StarCoder2-15B [40]	-	37.8	53.1	●	●	●
CodeQwen1.5-7B-Base [64]	-	45.7	60.2	○	-	-
CodeQwen1.5-7B-OctoPack	Publicly available	50.6	63.2	●	●	●
SelfCodeAlign-CQ-7B	Self-generated	67.1	65.2	●	●	●

HumanEval+ and MBPP+. HumanEval [8] and MBPP [3] are the two most widely-used benchmarks for function-level code generation. We use their test augmented versions, *i.e.,* HumanEval+ and MBPP+, with 80×/35× more test cases for rigorous evaluation [38].

As baselines, we consider a diverse set of state-of-the-art instruction-tuned models over various dimensions, including weight openness, data openness, transparency, and performance. Table 1 compares the pass@1 of the self-aligned SelfCodeAlign-CQ-7B against other baseline models on

HumanEval+ and MBPP+. Among those trained using a fully transparent pipeline without any proprietary data or distillation, SelfCodeAlign-CQ-7B stands out as the best LLM by drastically outperforming the base model, OctoCoder-16B, StarCoder2-15B, and CodeQwen1.5-7B-OctoPack. Meanwhile, compared to much larger models, SelfCodeAlign-CQ-7B outperforms Arctic, Command-R+, and Mixtral-8x7B-Instruct, while closely matching Mixtral-8x22B-instruct. Even compared to LLMs trained using proprietary data (*e.g.,* manually annotated), SelfCodeAlign-CQ-7B remains competitive, surpassing Gemini Pro, Mistral Large, and CodeLlama-70B-Instruct. Additionally, SelfCodeAlign-CQ-7B, fine-tuned on purely self-generated data, closely rivals models finetuned with distillation-based or non-transparent synthetic data.

LiveCodeBench. In subsequent evaluations, we benchmark our model against state-of-the-art open-source LLMs of similar sizes for a fair comparison. LiveCodeBench [21] is a benchmark for contamination-free evaluation. It features 400 recent Python algorithm challenges from May 2023 to February 2024. These tasks are curated from online judge websites such as Codeforce and LeetCode, each with over 20 test cases on average. While LiveCodeBench is a holistic benchmark covering four problem types, we focus on the code generation task for assessing LLM function generation.

Table 2 reports the pass@1 results for problem subsets created after three specific start dates. It shows that SelfCodeAlign-CQ-7B consistently outperforms most baseline models and closely matches CodeQwen1.5-7B-Chat. In addition, moving the start date forward has minimal impact on the pass@1 of SelfCodeAlign-CQ-7B, indicating that our pipeline is less likely to suffer from contamination.

Table 2: Pass@1 (%) of LLMs on LiveCodeBench. Newer start dates imply lower contamination risk.

Model	Start date		
	2023-09-01	2023-07-01	2023-05-01
DeepSeek-Coder-6.7B-Instruct	19.2	20.8	21.6
CodeGemma-7B-IT	15.2	14.1	13.6
Llama-3-8B-Instruct	18.3	18.4	17.3
CodeQwen1.5-7B-Base	19.3	20.7	21.8
CodeQwen1.5-7B-Chat	**23.2**	**24.1**	**25.0**
OctoCoder-16B	12.6	11.2	9.8
StarCoder2-15B	14.5	14.7	15.4
CodeQwen1.5-7B-OctoPack	19.3	21.8	22.5
SelfCodeAlign-CQ-7B	**22.4**	**22.8**	**23.4**

EvoEval. To mitigate the impact of potential data contamination, EvoEval [76] includes 828 programming problems created by prompting GPT-4 to evolve original HumanEval tasks across 5 semantic-altering and 2 semantic-preserving benchmarks. Following the leaderboard of EvoEval, we use the 5 semantic-altering benchmarks, each of which has 100 problems.

Table 3 shows that SelfCodeAlign-CQ-7B achieves the best pass@1 score among all transparently finetuned models. Meanwhile, it also surpasses most open LLMs (except CodeQwen1.5-7B-Chat) trained on unknown instruction-tuning data.

Table 3: Pass@1 (%) of code LLMs on EvoEval.

Model	Average	Difficult	Creative	Subtle	Combine	Tool use
DeepSeek-Coder-6.7B-Instruct	41.4	40	37	61	18	51
CodeGemma-7B-IT	35.4	31	32	49	9	56
Llama-3-8B-Instruct	40.6	34	39	57	15	58
CodeQwen1.5-7B-Base	36.2	26	30	46	18	61
CodeQwen1.5-7B-Chat	**48.0**	39	38	71	31	61
OctoCoder-16B	30.6	19	26	43	11	54
StarCoder2-15B	25.8	16	19	41	5	48
CodeQwen1.5-7B-OctoPack	42.2	35	36	59	22	59
SelfCodeAlign-CQ-7B	**43.6**	33	40	60	20	65

EvalPerf. While the earlier benchmarks focus on code correctness, we use EvalPerf [39] to evaluate the efficiency of LLM-generated code. EvalPerf includes 118 performance-exercising tasks with computation-intensive test inputs to fully exercise the efficiency of LLM-generated code.

Since code efficiency only matters when the generated code is correct, in Table 4 we only evaluate baselines that can achieve a decent pass@1 (*i.e.,* over 50%) on HumanEval+. Specifically, we run EvalPerf by following its default settings: *(i)* Each model generates 100 samples per task at the temperature of 1.0; *(ii)* We evaluate the efficiency of up to 20 correct samples per model for tasks where it can at least generate 10 passing samples; and *(iii)* Finally we rank the models based on their win rates, where each model pair compares their differential performance score (DPS) over the common set of passing tasks. Notably, DPS is a LeetCode-inspired metric that indicates the overall efficiency ranking of submissions. For example, Table 4 shows that SelfCodeAlign-CQ-7B achieves a DPS of 79.9, indicating that its correctly generated solutions can overall outperform or match the efficiency 79.9% of reference solutions across various efficiency levels.

Table 4 shows that SelfCodeAlign-CQ-7B ranks second among the evaluated models of comparable size. Specifically, SelfCodeAlign-CQ-7B is only next to DeepSeek-Coder-6.7B-Instruct whose training data is not disclosed. Surprisingly, the efficiency of SelfCodeAlign-CQ-7B-generated code surpasses many recent open models trained using private data, including the latest Llama-3.1-8B-Instruct.

Table 4: Ranking of model code efficiency based on the EvalPerf win rates, which are computed over the common set of passing tasks for each model pair. Each model generates 100 samples per task at a temperature 1.0. To exemplify differential performance score (DPS) with SelfCodeAlign-CQ-7B, it means its generations if correct can match the efficiency of 79.9% LLM samples.

Model	DPS (%)	pass@1 (%)	Win-rate (%)
DeepSeek-Coder-6.7B-Instruct	83.6	73.6	63.9
Llama-3.1-8B-Instruct	80.9	64.3	52.1
Llama-3-8B-Instruct	77.0	43.7	51.5
CodeQwen1.5-7B-Chat	80.7	74.1	51.2
CodeQwen1.5-7B-OctoPack	74.0	49.1	26.9
SelfCodeAlign-CQ-7B	79.9	65.2	54.0

3.2 Class-level Code Generation

We evaluate code LLMs on class-level code generation using ClassEval [13], a collection of 100 class-level Python code generation tasks, covering 100 classes and 410 methods, with an average of 33 tests per class and 8 tests per method.

Following the ClassEval paper [13], we set the maximum model context size as 2048 tokens and report the best class-level pass@1 (and corresponding method-level pass@1) of each model among three generation strategies: (i) *Holistic Generation*: generating the entire class given a class skeleton, (ii) *Incremental Generation*: generating class methods iteratively by putting earlier generated methods in the prompt, and (iii) *Compositional Generation*: generating each class method independently without looking at other methods. Specifically, class-level pass@1 in Table 5 refers to the pass rate of generated classes given *both* the method- and class-level tests. In contrast, method-level pass@1 is computed by *only* checking if the generated methods can pass the method-level tests. Table 5 shows, in terms of class-level performance, SelfCodeAlign-CQ-7B is the best transparently finetuned model, surpassing the second-best transparent model (*i.e.,* CodeQwen1.5-7B-OctoPack) by 28%, while performing no worse than those using unknown or proprietary instruction-tuning data. For method generation, SelfCodeAlign-CQ-7B also stands out in transparently finetuned models.

3.3 Data Science Programming

DS-1000 [30] is a benchmark of 1000 realistic data science challenges across 7 popular Python data science libraries. In DS-1000, a model must complete a partial code snippet to solve the problem. The solution is then evaluated through test execution. Table 6 shows that SelfCodeAlign-CQ-7B, despite being trained on limited data science code, stands out as the best in the transparent model category, while remaining competitive among the other evaluated baselines.

Table 5: Pass@1 (%) of code LLMs on ClassEval using greedy decoding.

Model	Class-level	Method-level
DeepSeek-Coder-6.7B-Instruct	**27.0**	**57.2**
CodeGemma-7B-IT	21.0	44.8
Llama-3-8B-Instruct	23.0	52.4
CodeQwen1.5-7B-Base	23.0	52.8
CodeQwen1.5-7B-Chat	**27.0**	54.6
OctoCoder-16B	19.0	38.0
StarCoder2-15B	9.0	24.9
CodeQwen1.5-7B-OctoPack	21.0	45.2
SelfCodeAlign-CQ-7B	**27.0**	**52.6**

Table 6: Pass@1 (%) on DS-1000 with temperature 0.2 and top-p 0.95 over 40 samples, following the same hyperparameter setting used in StarCoder2 [40].

Model	Avg.	Pandas	NumPy	Matplotlib	TensorFlow	SciPy	Sklearn	PyTorch
DeepSeek-Coder-6.7B-Instruct	44.6	34.0	51.1	58.4	45.9	34.2	45.8	50.6
CodeGemma-7B-IT	30.8	21.9	34.4	54.7	25.1	21.8	22.6	34.5
Llama-3-8B-Instruct	31.1	21.5	33.1	51.9	34.4	25.2	23.8	37.2
CodeQwen1.5-7B-Base	32.4	21.6	35.9	56.7	28.8	28.2	30.9	23.8
CodeQwen1.5-7B-Chat	**47.1**	34.4	51.7	67.2	46.0	38.9	47.9	52.8
OctoCoder-16B	28.3	13.1	34.0	53.8	22.4	22.8	30.0	25.9
StarCoder2-15B	38.9	26.2	45.8	61.4	38.1	36.0	40.5	22.5
CodeQwen1.5-7B-OctoPack	38.2	26.7	42.6	61.8	37.7	32.7	36.6	31.4
SelfCodeAlign-CQ-7B	**39.1**	28.2	42.6	57.2	38.3	35.6	42.8	33.3

3.4 Code Editing

We further evaluate LLMs on code editing tasks using the CanItEdit benchmark [6], comprised of 210 code editing tasks from three change kinds (70 tasks each): corrective (fixing bugs), adaptive (adding new features), and perfective (improving existing features). The tasks are evaluated based on the correctness of the generated code changes, according to a set of hidden test cases. For each task, the model is given as input the original code snippet and a natural-language instruction describing the desired code change; then it is expected to produce an updated code snippet that satisfies the instruction. We follow the setting from the original benchmark [6] to generate 20 completions per task at a temperature of 0.2. Table 7 reports the pass@1 for each change kind and the average pass@1 across all tasks. Despite not being specifically tuned for code editing, SelfCodeAlign-CQ-7B exhibits strong performance on CanItEdit, achieving a pass@1 of 39.0%, outperforming all other models except CodeQwen1.5-Chat, whose instruction tuning details are not disclosed.

Table 7: Pass@1 (%) of code LLMs on CanItEdit.

Model	Average	Corrective	Adaptive	Perfective
DeepSeek-Coder-6.7B-Instruct	36.3	34.9	38.8	35.3
CodeGemma-7B-IT	34.2	30.9	39.3	32.5
Llama-3-8B-Instruct	36.0	34.9	39.1	34.0
CodeQwen1.5-7B-Base	38.4	34.7	45.6	34.9
CodeQwen1.5-7B-Chat	**39.9**	38.1	46.6	35.1
OctoCoder-16B	30.2	38.4	31.6	20.5
StarCoder2-15B	36.7	32.1	43.8	34.2
CodeQwen1.5-7B-OctoPack	36.5	36.9	40.6	31.9
SelfCodeAlign-CQ-7B	**39.0**	37.4	42.4	37.2

4 Component Analysis

In this section, we extensively study how different components contribute to the SelfCodeAlign pipeline. To make the comparison tractable, we fix a subset of seed code snippets by randomly sampling 37k examples from the 250k corpus and evaluate finetuned models on HumanEval+ [38].

4.1 Self-Alignment with Different Models

To assess whether SelfCodeAlign is generalizable and how performance varies with finetuning data generated by different models, we run the same data generation pipeline end to end with different LLMs. We include four diverse state-of-the-art model architectures and sizes ranging from 3B to 33B to observe how SelfCodeAlign performs across small, medium, and large-scale LLMs.

Table 8 shows the comparison and guides us to reach the following findings. Looking at the diagonal cells, SelfCodeAlign consistently improves the performance of the base models with varying sizes, from 3B to 33B. Comparing each diagonal cell and the cell immediately to its right (i.e., using base models with slightly better HumanEval+ performance as the teacher models), we can see that a base model may benefit more from self-generated data than a stronger teacher model, when they don't have a large performance gap. However, when the teacher model is clearly stronger, the base model learns better by distilling the teacher's knowledge. For example, StarCoder2-3B achieves higher pass@1 trained on its own data (35.4) compared to Llama-3-8B data (34.1), but when tuned with stronger models, StarCoder2-3B further improves (*e.g.,* 42.1 with DeepSeek-Coder-33B data). Also, the last row shows that a stronger model can still learn from a weaker model, but less effectively. We provide qualitative examples in Appendix D.2.

Table 8: HumanEval+ pass@1 when finetuning the base models on different data (37k seeds).

Base model (pass@1)	Data-generation model				
	StarCoder2-3B	Llama-3-8B	StarCoder2-15B	DeepSeek-Coder-33B	CodeQwen1.5-7B
StarCoder2-3B (27.4)	35.4	34.1	39.0	**42.1**	40.2
Llama-3-8B (29.3)	-	42.7	40.2	41.5	**43.3**
StarCoder2-15B (37.8)	-	-	55.5	53.0	**57.3**
DeepSeek-Coder-33B (44.5)	-	-	-	**65.9**	62.2
CodeQwen1.5-7B (45.7)	48.8	54.9	56.1	59.1	**65.2**

4.2 Effectiveness of Execution-based Filtering

The SelfCodeAlign pipeline samples multiple responses for an instruction and each response is equipped with self-generated test cases. Responses with failing tests are filtered out and each instruction will be paired with a randomly selected passing response. To answer the question of "to what extent is execution information helpful", in Table 9, we conduct 4 controlled experiments by varying how responses are selected while keeping the other components unchanged:

- **Random selection (all)**: pair each instruction with a random response without response filtering.
- **Random selection (subset)**: 15.6k subset of "Random selection (all)" for a consistent data amount.
- **Failures only**: pair each instruction with a failing response.
- **Passes only**: pair each instruction with a passing response.

Table 9: Pass@1 on HumanEval+ with different response selection strategies.

Selection strategy	Data size	Execution pass rate	Pass@1
Random selection (all)	27.7k	24.1%	61.6
Random selection (subset)	15.6k	24.2%	61.6
Failures only	15.6k	0%	57.9
Passes only	15.6k	100.0%	**65.2**

First, we can observe that random pairing performs worse than using only passing examples, both when data sizes are aligned and when they scale up by 1.8×. Meanwhile, the "Failure only" setting results in the worst performance where we deliberately use failing responses for each instruction. These results suggest the importance of execution filtering and code correctness for self-alignment.

4.3 Importance of Seed Selection and Concepts Generation

For instruction generation, SelfCodeAlign applies Self-OSS-Instruct that first selects high-quality seed code snippets, then mines code concepts from the seeds, and finally generates the instructions. To validate the usefulness of concept generation and high-quality seeds, we compare two variants of SelfCodeAlign in Table 10: 1) directly generating instructions from seeds, where the model produces an instruction based solely on a seed snippet, and 2) using the default pipeline except for the initial seeds, where random snippets are sampled from different code documents in The Stack V1.

Table 10: Pass@1 on HumanEval+ using different seeds and pipelines.

Source of seeds	Pipeline	Pass@1
Filtered functions	Seed $\rightarrow$ instruction	59.8
Random snippets	Seed $\rightarrow$ concepts $\rightarrow$ instruction	64.0
Filtered functions	Seed $\rightarrow$ concepts $\rightarrow$ instruction	**65.2**

It is shown that directly generating instructions from seeds leads to the poorest performance. This is because a direct generation from seeds requires the seed snippet to be presented in the context, whose format is not well represented in the wild and may not be in distribution for the model. The generated instructions will then be distracted and thus be of lower quality. Concept generation neutralizes this effect and produces more realistic and natural instructions. Using random snippets produces a more diverse but less coherent set of concepts, leading to slightly worse performance compared to using high-quality seeds. Appendices D.3 and D.4 illustrate some qualitative examples.

4.4 Comparing Self-Alignment to Distillation

Table 11: SelfCodeAlign versus distillation using CodeQwen1.5-7B as the base model.

Method	Dataset size	Teacher model	Execution filtering	Pass@1
Evol-Instruct	74k	GPT-3.5-Turbo	○	59.1
OSS-Instruct	74k	GPT-3.5-Turbo	○	61.6
Direct distillation	74k	GPT-4o	○	65.9
SelfCodeAlign	74k	CodeQwen1.5-7B	●	**67.1**

To compare self-alignment with distillation, we evaluate SelfCodeAlign against two state-of-the-art distillation methods for code instruction tuning: OSS-Instruct [72] and Code Evol-Instruct [65]. We use the official OSS-Instruct dataset. As the official implementation of Code Evol-Instruct is unavailable, we opt for the most popular open-source version [44] on Hugging Face. Both datasets are generated using GPT-3.5-Turbo [46] and we randomly select their subsets to match the 74k samples generated by SelfCodeAlign. Table 11 shows that SelfCodeAlign substantially outperforms both methods, indicating the strength and promising future of self-alignment for code. Additionally, SelfCodeAlign outperforms direct distillation, where we use the same set of SelfCodeAlign instructions but rely on GPT-4o [49] to generate each response at temperature 0. This suggests that weaker models, combined with more post-validation compute, can produce higher-quality responses.

5 Related Work

Instruction tuning for code. To build more powerful code assistants, pre-trained code models are fine-tuned over a small amount of high-quality instruction-response pairs that are either collected from real-world [43] or synthetically generated [7, 57, 41, 72]. This step is known as instruction tuning. OctoPack [43] compiles a large set of real-world Git commits which are partially used for code fine-tuning. Code Alpaca [7] applies SELF-INSTRUCT to the code domain, which prompts ChatGPT to generate synthetic instruction data for code. Similarly, the instruction data for CODELLAMA [57] includes coding problems generated by prompting LLAMA 2 [66] and solutions and tests by prompting base CODELLAMA. Code Evol-Instruct [41] uses harder programming challenges as instruction data to fine-tune more capable models. Specifically, Code Evol-Instruct prompts ChatGPT with heuristics to evolve existing instruction data to more challenging and complex ones. Besides data complexity, the widely-adopted [14, 62, 71] OSS-INSTRUCT [72] looks at the data *diversity* and *quality* dimension. Specifically, given a source code snippet, OSS-INSTRUCT prompts ChatGPT to get inspired and imagine potential instruction-response pairs, which inherit the diversity and quality of sampled code snippets. Besides instruction tuning, recent work on training code LLMs for performance improvement also explores multi-turn code generation [83], model merging [12], preference tuning [74, 36], and reinforcement learning [15]. Recently, various strong instruction-tuned code models have been released by major organizations [19, 64]. However, their instruction-tuning recipes (*e.g.,* data and strategies) are not fully disclosed. This lack of transparency underscores the need for fully transparent and permissive instruction-tuning methods to advance the field.

Self-alignment. Self-alignment is an approach to instruction tuning that utilizes an LLM to learn from its own output without depending on an existing well-aligned teacher LLM. SELF-INSTRUCT [68] is one of the first endeavors that allow GPT-3 to improve itself by generating new instructions and responses for instruction-tuning using its in-context learning capability. SELF-ALIGN [61], based on in-context learning, utilizes topic-guided SELF-INSTRUCT for instruction generation and pre-defines principles to steer the LLM towards desired responses. These instruction-response pairs are used to fine-tune the base model, followed by a final refinement stage to ensure the model produces in-depth and detailed responses. Instruction backtranslation [35] offers an alternative self-alignment method by initially training a backward model to generate instructions from unlabeled web documents using limited seed data. It then iteratively produces new instructions from new web documents and selects high-quality data for self-training. Most code LLM work targets knowledge distillation. Haluptzok et al. [20] share a relevant idea to our work but only consider program puzzles specified in symbolic forms. This setting cannot be generalized to real-world tasks with natural language involved.

6 Limitations and Future Work

We limit our data generation within a $\sim$3000 window, skewing our distribution towards medium-sized samples. Therefore, generating and training on long-context instruction-response pairs can be a promising avenue [4]. Second, we gather several negative samples during response generation, which are currently filtered out. These negatives could be used in a reinforcement-learning loop to steer the model away from incorrect responses [31, 53]. Furthermore, the good responses are labeled by test execution, while the generated unit tests might be erroneous, calling for research to study and improve the generation of valid test cases. Finally, we plan to apply SelfCodeAlign to more challenging domains such as complex program generation [84] and agentic software engineering [26].

7 Conclusion

We introduce SelfCodeAlign, the first fully transparent and permissive pipeline for self-aligning code LLMs without extensive human annotations or distillation. SelfCodeAlign-CQ-7B, finetuned from CodeQwen1.5-7B using SelfCodeAlign, outperforms the 10$\times$ larger CodeLlama-70B-Instruct on HumanEval+ and consistently surpasses CodeQwen1.5 trained with OctoPack on all studied benchmarks. We evaluate SelfCodeAlign across various model sizes, illustrating that stronger base models benefit more from self-alignment than distillation. We also examine the effectiveness of different components in the pipeline, showing that SelfCodeAlign is better than GPT-3.5 and GPT-4o distillation. Overall, we demonstrate for the first time that a strong instruction-tuned code LLM can be created through self-alignment, without expensive human annotations or distillation.

Acknowledgements

We thank all the reviewers for their insightful comments and suggestions for our paper. This work was partially supported by NSF grants CCF-2131943, SES-2326173, and Kwai Inc, as well as API credits from the OpenAI Researcher Access Program.

References

[1] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590*, 2021.

[2] Anthropic. Terms of service, 7 2023. Accessed: August 17, 2023.

[3] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021.

[4] Y. Bai, X. Lv, J. Zhang, Y. He, J. Qi, L. Hou, J. Tang, Y. Dong, and J. Li. Longalign: A recipe for long context alignment of large language models, 2024.

[5] F. Cassano, J. Gouwar, F. Lucchetti, C. Schlesinger, A. Freeman, C. J. Anderson, M. Q. Feldman, M. Greenberg, A. Jangda, and A. Guha. Knowledge transfer from high-resource to low-resource programming languages for Code LLMs, 2024.

[6] F. Cassano, L. Li, A. Sethi, N. Shinn, A. Brennan-Jones, A. Lozhkov, C. J. Anderson, and A. Guha. Can It Edit? Evaluating the Ability of Large Language Models to Follow Code Editing Instructions. In *The First International Workshop on Large Language Model for Code*, 2024.

[7] S. Chaudhary. Code alpaca: An instruction-following llama model for code generation. `https://github.com/sahil280114/codealpaca`, 2023.

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code, 2021.

[9] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve, et al. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*, 2023.

[10] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models, 2023.

[11] Y. Ding, H. Ding, S. Wang, Q. Sun, V. Kumar, and Z. Wang. Horizon-length prediction: Advancing fill-in-the-middle capabilities for code generation with lookahead planning. *arXiv preprint arXiv:2410.03103*, 2024.

[12] Y. Ding, J. Liu, Y. Wei, and L. Zhang. XFT: Unlocking the power of code instruction tuning by simply merging upcycled mixture-of-experts. In L.-W. Ku, A. Martins, and V. Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12941–12955, Bangkok, Thailand, Aug. 2024. Association for Computational Linguistics.

[13] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*, 2023.

[14] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, A. Goyal, A. Hartshorn, A. Yang, A. Mitra, A. Sravankumar, A. Korenev, A. Hinsvark, A. Rao, A. Zhang, A. Rodriguez, A. Gregerson, A. Spataru, B. Roziere, B. Biron, B. Tang, B. Chern, C. Caucheteux, C. Nayak, C. Bi, C. Marra, C. McConnell, C. Keller, C. Touret, C. Wu, C. Wong, C. C. Ferrer, C. Nikolaidis, D. Allonsius, D. Song, D. Pintz, D. Livshits, D. Esiobu, D. Choudhary, D. Mahajan, D. Garcia-Olano, D. Perino, D. Hupkes, E. Lakomkin, E. AlBadawy, E. Lobanova, E. Dinan, E. M. Smith, F. Radenovic, F. Zhang, G. Synnaeve, G. Lee, G. L. Anderson, G. Nail, G. Mialon, G. Pang, G. Cucurell, H. Nguyen, H. Korevaar, H. Xu, H. Touvron, I. Zarov, I. A. Ibarra, I. Kloumann, I. Misra, I. Evtimov, J. Copet, J. Lee, J. Geffert, J. Vranes, J. Park, J. Mahadeokar, J. Shah, J. van der Linde, J. Billock, J. Hong, J. Lee, J. Fu, J. Chi, J. Huang, J. Liu, J. Wang, J. Yu, J. Bitton, J. Spisak, J. Park, J. Rocca, J. Johnstun, J. Saxe, J. Jia, K. V. Alwala, K. Upasani, K. Plawiak, K. Li, K. Heafield, K. Stone, K. El-Arini, K. Iyer, K. Malik, K. Chiu, K. Bhalla, L. Rantala-Yeary, L. van der

Maaten, L. Chen, L. Tan, L. Jenkins, L. Martin, L. Madaan, L. Malo, L. Blecher, L. Landzaat, L. de Oliveira, M. Muzzi, M. Pasupuleti, M. Singh, M. Paluri, M. Kardas, M. Oldham, M. Rita, M. Pavlova, M. Kambadur, M. Lewis, M. Si, M. K. Singh, M. Hassan, N. Goyal, N. Torabi, N. Bashlykov, N. Bogoychev, N. Chatterji, O. Duchenne, O. Çelebi, P. Alrassy, P. Zhang, P. Li, P. Vasic, P. Weng, P. Bhargava, P. Dubal, P. Krishnan, P. S. Koura, P. Xu, Q. He, Q. Dong, R. Srinivasan, R. Ganapathy, R. Calderer, R. S. Cabral, R. Stojnic, R. Raileanu, R. Girdhar, R. Patel, R. Sauvestre, R. Polidoro, R. Sumbaly, R. Taylor, R. Silva, R. Hou, R. Wang, S. Hosseini, S. Chennabasappa, S. Singh, S. Bell, S. S. Kim, S. Edunov, S. Nie, S. Narang, S. Raparthy, S. Shen, S. Wan, S. Bhosale, S. Zhang, S. Vandenhende, S. Batra, S. Whitman, S. Sootla, S. Collot, S. Gururangan, S. Borodinsky, T. Herman, T. Fowler, T. Sheasha, T. Georgiou, T. Scialom, T. Speckbacher, T. Mihaylov, T. Xiao, U. Karn, V. Goswami, V. Gupta, V. Ramanathan, V. Kerkez, V. Gonguet, V. Do, V. Vogeti, V. Petrovic, W. Chu, W. Xiong, W. Fu, W. Meers, X. Martinet, X. Wang, X. E. Tan, X. Xie, X. Jia, X. Wang, Y. Goldschlag, Y. Gaur, Y. Babaei, Y. Wen, Y. Song, Y. Zhang, Y. Li, Y. Mao, Z. D. Coudert, Z. Yan, Z. Chen, Z. Papakipos, A. Singh, A. Grattafiori, A. Jain, A. Kelsey, A. Shajnfeld, A. Gangidi, A. Victoria, A. Goldstand, A. Menon, A. Sharma, A. Boesenberg, A. Vaughan, A. Baevski, A. Feinstein, A. Kallet, A. Sangani, A. Yunus, A. Lupu, A. Alvarado, A. Caples, A. Gu, A. Ho, A. Poulton, A. Ryan, A. Ramchandani, A. Franco, A. Saraf, A. Chowdhury, A. Gabriel, A. Bharambe, A. Eisenman, A. Yazdan, B. James, B. Maurer, B. Leonhardi, B. Huang, B. Loyd, B. D. Paola, B. Paranjape, B. Liu, B. Wu, B. Ni, B. Hancock, B. Wasti, B. Spence, B. Stojkovic, B. Gamido, B. Montalvo, C. Parker, C. Burton, C. Mejia, C. Wang, C. Kim, C. Zhou, C. Hu, C.-H. Chu, C. Cai, C. Tindal, C. Feichtenhofer, D. Civin, D. Beaty, D. Kreymer, D. Li, D. Wyatt, D. Adkins, D. Xu, D. Testuggine, D. David, D. Parikh, D. Liskovich, D. Foss, D. Wang, D. Le, D. Holland, E. Dowling, E. Jamil, E. Montgomery, E. Presani, E. Hahn, E. Wood, E. Brinkman, E. Arcaute, E. Dunbar, E. Smothers, F. Sun, F. Kreuk, F. Tian, F. Ozgenel, F. Caggioni, F. Guzmán, F. Kanayet, F. Seide, G. M. Florez, G. Schwarz, G. Badeer, G. Swee, G. Halpern, G. Thattai, G. Herman, G. Sizov, Guangyi, Zhang, G. Lakshminarayanan, H. Shojanazeri, H. Zou, H. Wang, H. Zha, H. Habeeb, H. Rudolph, H. Suk, H. Aspegren, H. Goldman, I. Damlaj, I. Molybog, I. Tufanov, I.-E. Veliche, I. Gat, J. Weissman, J. Geboski, J. Kohli, J. Asher, J.-B. Gaya, J. Marcus, J. Tang, J. Chan, J. Zhen, J. Reizenstein, J. Teboul, J. Zhong, J. Jin, J. Yang, J. Cummings, J. Carvill, J. Shepard, J. McPhie, J. Torres, J. Ginsburg, J. Wang, K. Wu, K. H. U, K. Saxena, K. Prasad, K. Khandelwal, K. Zand, K. Matosich, K. Veeraraghavan, K. Michelena, K. Li, K. Huang, K. Chawla, K. Lakhotia, K. Huang, L. Chen, L. Garg, L. A, L. Silva, L. Bell, L. Zhang, L. Guo, L. Yu, L. Moshkovich, L. Wehrstedt, M. Khabsa, M. Avalani, M. Bhatt, M. Tsimpoukelli, M. Mankus, M. Hasson, M. Lennie, M. Reso, M. Groshev, M. Naumov, M. Lathi, M. Keneally, M. L. Seltzer, M. Valko, M. Restrepo, M. Patel, M. Vyatskov, M. Samvelyan, M. Clark, M. Macey, M. Wang, M. J. Hermoso, M. Metanat, M. Rastegari, M. Bansal, N. Santhanam, N. Parks, N. White, N. Bawa, N. Singhal, N. Egebo, N. Usunier, N. P. Laptev, N. Dong, N. Zhang, N. Cheng, O. Chernoguz, O. Hart, O. Salpekar, O. Kalinli, P. Kent, P. Parekh, P. Saab, P. Balaji, P. Rittner, P. Bontrager, P. Roux, P. Dollar, P. Zvyagina, P. Ratanchandani, P. Yuvraj, Q. Liang, R. Alao, R. Rodriguez, R. Ayub, R. Murthy, R. Nayani, R. Mitra, R. Li, R. Hogan, R. Battey, R. Wang, R. Maheswari, R. Howes, R. Rinott, S. J. Bondu, S. Datta, S. Chugh, S. Hunt, S. Dhillon, S. Sidorov, S. Pan, S. Verma, S. Yamamoto, S. Ramaswamy, S. Lindsay, S. Lindsay, S. Feng, S. Lin, S. C. Zha, S. Shankar, S. Zhang, S. Zhang, S. Wang, S. Agarwal, S. Sajuyigbe, S. Chintala, S. Max, S. Chen, S. Kehoe, S. Satterfield, S. Govindaprasad, S. Gupta, S. Cho, S. Virk, S. Subramanian, S. Choudhury, S. Goldman, T. Remez, T. Glaser, T. Best, T. Kohler, T. Robinson, T. Li, T. Zhang, T. Matthews, T. Chou, T. Shaked, V. Vontimitta, V. Ajayi, V. Montanez, V. Mohan, V. S. Kumar, V. Mangla, V. Albiero, V. Ionescu, V. Poenaru, V. T. Mihailescu, V. Ivanov, W. Li, W. Wang, W. Jiang, W. Bouaziz, W. Constable, X. Tang, X. Wang, X. Wu, X. Wang, X. Xia, X. Wu, X. Gao, Y. Chen, Y. Hu, Y. Jia, Y. Qi, Y. Li, Y. Zhang, Y. Zhang, Y. Adi, Y. Nam, Yu, Wang, Y. Hao, Y. Qian, Y. He, Z. Rait, Z. DeVito, Z. Rosnbrick, Z. Wen, Z. Yang, and Z. Zhao. The llama 3 herd of models, 2024.

[15] J. Gehring, K. Zheng, J. Copet, V. Mella, T. Cohen, and G. Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning, 2024.

[16] A. Gomez. Introducing command r+: A scalable llm built for business, April 4 2024. Accessed: 2024-05-22.

[17] Google. Generative ai terms of service, 8 2023. Accessed: August 17, 2023.

[18] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. D. Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi, A. Salim, S. Shah, H. S. Behl, X. Wang, S. Bubeck, R. Eldan, A. T. Kalai, Y. T. Lee, and Y. Li. Textbooks are all you need, 2023.

[19] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

[20] P. Haluptzok, M. Bowers, and A. T. Kalai. Language models can teach themselves to program better. In *The Eleventh International Conference on Learning Representations*, 2023.

[21] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[22] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed. Mistral 7b, 2023.

[23] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. de las Casas, E. B. Hanna, F. Bressand, G. Lengyel, G. Bour, G. Lample, L. R. Lavaud, L. Saulnier, M.-A. Lachaux, P. Stock, S. Subramanian, S. Yang, S. Antoniak, T. L. Scao, T. Gervet, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed. Mixtral of experts, 2024.

[24] N. Jiang, K. Liu, T. Lutellier, and L. Tan. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1430–1442. IEEE, 2023.

[25] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

[26] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2023.

[27] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656, 2023.

[28] D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries. The stack: 3 tb of permissively licensed source code, 2022.

[29] A. Köpf, Y. Kilcher, D. von Rütte, S. Anagnostidis, Z. R. Tam, K. Stevens, A. Barhoum, D. M. Nguyen, O. Stanley, R. Nagyfi, S. ES, S. Suri, D. A. Glushkov, A. V. Dantuluri, A. Maguire, C. Schuhmann, H. Nguyen, and A. J. Mattick. Openassistant conversations - democratizing large language model alignment. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023.

[30] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, S. W. tau Yih, D. Fried, S. Wang, and T. Yu. Ds-1000: A natural and reliable benchmark for data science code generation, 2022.

[31] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[32] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931. IEEE, 2023.

[33] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries. Starcoder: may the source be with you!, 2023.

[34] X. Li, P. Yu, C. Zhou, T. Schick, O. Levy, L. Zettlemoyer, J. E. Weston, and M. Lewis. Self-alignment with instruction backtranslation. In *The Twelfth International Conference on Learning Representations*, 2024.

[35] X. Li, P. Yu, C. Zhou, T. Schick, O. Levy, L. Zettlemoyer, J. E. Weston, and M. Lewis. Self-alignment with instruction backtranslation. In *The Twelfth International Conference on Learning Representations*, 2024.

[36] J. Liu, T. Nguyen, M. Shang, H. Ding, X. Li, Y. Yu, V. Kumar, and Z. Wang. Learning code preference via synthetic evolution. *arXiv preprint arXiv:2410.03837*, 2024.

[37] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024.

[38] J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[39] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. Zhang. Evaluating language models for efficient code generation. In *First Conference on Language Modeling*, 2024.

[40] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

[41] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang. Wizardcoder: Empowering code large language models with evol-instruct, 2023.

[42] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[43] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. von Werra, and S. Longpre. Octopack: Instruction tuning code large language models, 2023.

[44] nickrosh. Open Source Implementation of Evol-Instruct-Code. https://huggingface.co/datasets/nickrosh/Evol-Instruct-Code-80k-v1, 2023.

[45] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2023.

[46] OpenAI. Chatgpt: Optimizing language models for dialogue. https://openai.com/blog/chatgpt/, 2022.

[47] OpenAI. Gpt-4 technical report, 2023.

[48] OpenAI. Terms of service, 3 2023. Accessed: August 17, 2023.

[49] OpenAI. Gpt-4o system card. 2024.

[50] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe. Training language models to follow instructions with human feedback, 2022.

[51] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand. Understanding the effectiveness of large language models in code translation. *arXiv preprint arXiv:2308.03109*, 2023.

[52] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training. 2018.

[53] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn. Direct preference optimization: Your language model is secretly a reward model. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[54] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[55] S. A. Research. Snowflake arctic: The best llm for enterprise ai — efficiently intelligent, truly open, April 24 2024. Accessed: 2024-05-22.

[56] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample. Unsupervised translation of programming languages. *Advances in neural information processing systems*, 33:20601–20611, 2020.

[57] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve. Code llama: Open foundation models for code, 2023.

[58] N. Shazeer and M. Stern. Adafactor: Adaptive learning rates with sublinear memory cost, 2018.

[59] A. Shypula, A. Madaan, Y. Zeng, U. Alon, J. Gardner, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*, 2023.

[60] Z. Sun, Y. Shen, H. Zhang, Q. Zhou, Z. Chen, D. D. Cox, Y. Yang, and C. Gan. SALMON: Self-alignment with instructable reward models. In *The Twelfth International Conference on Learning Representations*, 2024.

[61] Z. Sun, Y. Shen, Q. Zhou, H. Zhang, Z. Chen, D. D. Cox, Y. Yang, and C. Gan. Principle-driven self-alignment of language models from scratch with minimal human supervision. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[62] C. Team, H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo, A. Hu, C. A. Choquette-Choo, J. Shen, J. Kelley, K. Bansal, L. Vilnis, M. Wirth, P. Michel, P. Choy, P. Joshi, R. Kumar, S. Hashmi, S. Agrawal, Z. Gong, J. Fine, T. Warkentin, A. J. Hartman, B. Ni, K. Korevec, K. Schaefer, and S. Huffman. Codegemma: Open code models based on gemma, 2024.

[63] G. Team. Gemini: A family of highly capable multimodal models, 2024.

[64] Q. Team. Code with codeqwen1.5, April 16 2024. Accessed: 2024-05-20.

[65] theblackcat102. The evolved code alpaca dataset. `https://huggingface.co/datasets/theblackcat102/evol-codealpaca-v1`, 2023.

[66] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[67] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig. Openhands: An open platform for ai software developers as generalist agents, 2024.

[68] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In A. Rogers, J. Boyd-Graber, and N. Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, Toronto, Canada, July 2023. Association for Computational Linguistics.

[69] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi. Codet5+: Open code large language models for code understanding and generation, 2023.

[70] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, Nov. 2021. Association for Computational Linguistics.

[71] Y. Wei, H. Han, and R. Samdani. Arctic-snowcoder: Demystifying high-quality data in code pretraining. *arXiv preprint arXiv:2409.02326*, 2024.

[72] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, 2023.

[73] Y. Wei, C. S. Xia, and L. Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 172–184, New York, NY, USA, 2023. Association for Computing Machinery.

[74] M. Weyssow, A. Kamanda, and H. Sahraoui. Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences. *arXiv preprint arXiv:2403.09032*, 2024.

[75] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint*, 2024.

[76] C. S. Xia, Y. Deng, and L. Zhang. Top leaderboard ranking= top coding proficiency, always? evoeval: Evolving coding benchmarks via llm. *arXiv preprint arXiv:2403.19114*, 2024.

[77] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang. Universal fuzzing via large language models, 2023.

[78] C. S. Xia, Y. Wei, and L. Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494, 2023.

[79] C. S. Xia and L. Zhang. Less training, more repairing please: Revisiting automated program repair via zero-shot learning, 2022.

[80] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024.

[81] Z. Yu, X. Zhang, N. Shang, Y. Huang, C. Xu, Y. Zhao, W. Hu, and Q. Yin. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation, 2024.

[82] W. Yuan, R. Y. Pang, K. Cho, X. Li, S. Sukhbaatar, J. Xu, and J. Weston. Self-rewarding language models, 2024.

[83] T. Zheng, G. Zhang, T. Shen, X. Liu, B. Y. Lin, J. Fu, W. Chen, and X. Yue. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658*, 2024.

[84] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, S. Brunner, C. Gong, T. Hoang, A. R. Zebaze, X. Hong, W.-D. Li, J. Kaddour, M. Xu, Z. Zhang, P. Yadav, N. Jain, A. Gu, Z. Cheng, J. Liu, Q. Liu, Z. Wang, D. Lo, B. Hui, N. Muennighoff, D. Fried, X. Du, H. de Vries, and L. V. Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions, 2024.