Survey of code generation and feedback loops and reproduction of Self-Refine

Akshay Badagabettu, Sai Sravan Yarlagadda, Wenjin Fu

{abadagab, saisravy, wenjinf}@andrew.cmu.edu

Abstract

For the research project, we plan to work on building an NLP system capable of generating CAD models using natural language queries. Since CAD design is an iterative process, achieving the best CAD design in a single attempt is uncommon. In this assignment, our primary focus was on surveying language models specifically used for code generation problems and exploring methods to provide feedback to the model to improve upon its previous generations. After conducting a literature survey, we found that the paper "SELF-REFINE: Iterative Refinement with Self-Feedback" most closely resembles the feedback loop in our architecture. We have re-implemented the code optimization task of this paper. The code used can be found in our GitHub [1]

1 Survey of code generation

Recent advances in language modeling have been propelled by the introduction of pre-trained Transformer models (Vaswani et al., 2017), such as BERT (Devlin et al., 2018) and GPT (Radford et al., 2018). As these large-scale language models (LLM) expanded to encompass hundreds of billions of parameters, exhibiting preliminary indications of artificial intelligence, their utility has expanded beyond mere text processing. Notably, large language models like ChatGPT (Ray, 2023) have demonstrated exceptional capabilities in code generation. This progress has given rise to specialized LLMs focused on software engineering, known as Code LLMs. Typically, these Code LLMs evolve from general-purpose LLMs through a process of fine-tuning, with their effectiveness being dependent on the foundational LLMs. This part of the literature review undertakes an exhaustive examination of both Code LLMs and general LLMs, aiming to resolve three key inquiries: (1) What are the prevalent benchmark datasets for evaluating code generation in LLMs? (2) In tasks related to code generation, do Code LLMs surpass the performance of general LLMs? (3) What prompting strategies have been identified in leading LLMs for code generation?

1.1 Methodology

To offer a comprehensive overview of current practices and benchmarks in the domain of LLM-facilitated code generation, we first identify prevalent benchmark datasets for evaluating code generation in LLMs. We conducted a detailed survey of public repositories and leaderboards (Papers with Code(Roy et al., 2020) and Hugging Face (Jain, 2022)) to identify benchmark datasets widely recognized and used in evaluating LLM's code generation. Next, using the identified benchmarks, we conduct a comparative analysis between general LLMs and Code LLMs on their code generation performance to discover whether Code LLMs demonstrate superior performance overall than general LLMs. Furthermore, with an exhaustive review of different LLMs' performance on leader boards, we identify LLMs that are more proficient in code generation tasks compared to others. This will include an analysis of both general and code LLMs to determine which models consistently outperform others across a range of benchmarks and tasks. Finally, by examining each of the top-performing LLM research papers, we uncover common prompting strategies and how they affect LLMs to achieve superior code generation.

1.2 Common benchmark datasets and Evaluation Metrics for LLMs in Code Generation

A diverse array of benchmark datasets is available for assessing the code generation capabilities of large language models (LLMs). Among these, HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), APPS (Hendrycks et al., 2021), CoNaLa

[1] https://github.com/akshay140601/Self-Refine-Reimplementation

(Yin et al., 2018), and CodeContests (Li et al., 2022b) are frequently utilized. Among these, HumanEval and MBPP are some of the most widely recognized benchmarks in the examination of LLMs' code generation proficiency. It stands out by its collection of 164 Python programming challenges, each comprising a detailed function description, signature, and illustrative test cases as assertions. LLMs are tasked with crafting complete functions to adhere to these prompts, aiming to successfully fulfill all test case requirements. Given code snippets with the same functionality may vary significantly in lexical presentation, instead of calculating lexical similarity as traditional metrics, many benchmark datasets, like HumanEval, focus on functional correctness in code generation. Researchers developed metrics like pass@k, initially introduced by (Kulal et al., 2019) and later refined by OpenAI (Chen et al., 2021). This metric offers an unbiased estimate of a model's likelihood to successfully pass all unit tests for a program from any k-generated attempts. An extension of this is passn@k, which restricts the number of submissions to n yet permits selecting the most promising solutions based on unit tests from the k attempts, enhancing the precision of assessing a model's code generation effectiveness.

MBPP (Austin et al., 2021), standing for Massively Bugs and Performance Problems, also serves as a critical benchmark for assessing LLMs' code generation skills. This benchmark includes numerous code examples containing errors and performance issues and challenging models to produce appropriate fixes. It tests models' capabilities in detecting and fixing software faults and performance issues, thus evaluating their effectiveness and resilience in real-world software engineering contexts.

1.3 Code LLMs and General LLMs which Better

Many researchers are dedicated to assessing and enhancing large language models' (LLMs) efficacy in software engineering roles, yet skepticism exists regarding their current capabilities in this area (Li et al., 2023b)(Pearce et al., 2023). This skepticism often stems from the challenges LLMs face in grasping the nuances of code structure and semantics, attributed to a lack of specific software engineering knowledge. Despite these challenges, targeted fine-tuning and additional training can improve LLMs' performance for software engineering applications, leading to the creation of specialized Code LLMs, such as StarCoder (Li et al., 2023a) and Code Llama (Roziere et al., 2023). With these advancements, and considering the impressive capabilities of general LLMs like GPT-4 (Achiam et al., 2023) in code generation, it becomes crucial to evaluate whether specialized Code LLMs indeed surpass general models in these tasks.

Model	Size	HumanEval	MBP
Llama 2	70B	29.90%	45.00%
Code Llama	70B	53.00%	62.40%
Code Llama - Instruct	70B	67.80%	62.20%
Code Llama - Python	70B	57.30%	65.60%
Code Llama - Python	34B	41.50%	57.00%
Code Llama - Python	7B	38.40%	47.60%
CodeFuse	34B	74.40%	-
Palm	540B	26.20%	36.80%
Palm - Coder	540B	36.00%	47.00%
StarCoder Base	15.5B	30.40%	49.00%
StarCoder Python	15.5B	33.60%	52.70%
StarCoder Prompted	15.5B	40.80%	49.50%
GPT-4	-	67/82%	-
GPT-3.5 (ChatGPT)	-	48.10%	52.20%

Figure 1: Pass@1 performance of Code LLMs and general LLMs performance on HumanEval and MBPP

Therefore, we organized several state-of-the-art large language models' performance results from the collected papers on HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) to gain some insights on the difference in the performance of Code LLMs and general LLMs, resulting in Figure 1. The introduction of Codex by Chen et al. (2021b) (Chen et al., 2021) marked the beginning era of LLMs for coding. Subsequent research has seen LLMs further tailored for code generation through additional pre-training efforts. For example, (Touvron et al., 2023) developed LLama 2 into Code LLaMA (Roziere et al., 2023) by training it on over 500 billion code tokens, achieving a passing rate of 53.0% and 62.4% on HumanEval and MBPP scores, much better than the original LLama 2. Two variants of Code LLaMA were further refined: Code LLaMA - Python (Roziere et al., 2023) and Code LLaMA - Instruct (Roziere et al., 2023). The former specifically fine-tuned on 100 billion Python code tokens, significantly outperformed the base model in code-related tasks despite its smaller size of 34B, showing lesser performance than the base model. Code LLaMA - Instruct, tailored through instruction fine-tuning, informing the model expected output given the input, even without further pre-trained for solving code-specific tasks, outperforming both the base model and its Python variant. (Liu et al., 2023)

introduced CodeFuse-CodeLLaMA via multi-task fine-tuning (MFT), achieving a 74.4 pass@1 rate on HumanEval, surpassing even GPT-4's performance according to OpenAI. Additional training on code-specific data sets has also led to the development of PalM-Coder (Chung et al., 2022) and StarCoder-Python (Li et al., 2023a), both showing superior performance to general LLMs. Despite this, GPT-4's (Achiam et al., 2023) (OpenAI, 2023) remarkably achieved a score of 67.0 (initially reported as 82 by (Bubeck et al., 2023)), maintaining a lead over many specialized or instruction-finetuned models for code until recently. Drawing from the aforementioned results, we arrive at these conclusions: 1. LLMs performance can be influenced by a number of model parameters; models with more parameters perform better than the same base model. 2. A model with fine-tuning for code-specific tasks may enhance its performance over the base model 3. Within the same parameter sizes, LLMs designed for code often surpass the performance of general-purpose base LLMs.

1.4 Common Prompting Strategies to Improve Code Generation

As shown in the previous section, with better prompting engineers, the base model like Code LLaMA - Instruct (Roziere et al., 2023) may outperform both the base model Code LLaMA and the model designed for handling coding tasks. (Hou and Ji, 2024) thoroughly explored six distinct prompt strategies to evaluate the performance of LLMs in solving programming tasks. The strategies included: 1. repeated prompt: providing the full programming task and example test case of the GPT-4 repeatedly, ignoring any error messages. 2. repeated prompt without example: similar to the first without the example test cases, to see if excluding them affects the LLM's performance. 3. Multiway Prompt: asking the LLM to generate five different solutions to the programming task at once. 4. feedback prompt: the LLM receives the programming task, and upon failure, it gets the error message as input for the improvement. 5. Feedback CI prompt: Similar to the feedback prompt, it uses a code interpreter (CI) to evaluate example test cases and improve solutions based on feedback from leetCode's error messages. 6. Python3 Translation: LLM translates Python3 code into Java, JavaScript, and C++, using feedback CI prompt strategy.

The effectiveness of GPT-4 was measured against 100 programming challenges from LeetCode, allowing up to five attempts per strategy for each task. The evaluation recorded both initial attempt success rates and success within five attempts. Their finding reveals that: 1. success rates for the repeated prompt strategy, including example cases, reached 86%, 58%, and 15% for easy, medium, and hard tasks, respectively, aligning with prior research and underscoring consistency across studies. 2. The code interpreter feedback strategy emerged as the most effective, markedly improving success rates across various levels of task difficulty compared to the repeated prompt strategy. 3. (Hou and Ji, 2024) study confirmed that GPT-4's efficiency is less affected by programming language differences than by the applied prompt strategy, with the code interpreter feedback strategy leading to consistent performance across languages. The results indicate that selecting an appropriate prompt strategy critically influences an LLM's ability to code, with the code interpreter feedback strategy significantly boosting GPT-4's efficacy in tackling tasks of differing complexities.

2 Survey of feedback loops

A common approach humans take to design CAD models involves creating an initial iteration and then refining that iteration by examining the design and identifying its flaws. For example, in the process of designing the CAD model of a water bottle, one would begin by creating a basic design of the bottle. Subsequently, upon reviewing the design, one may opt to modify and redesign if necessary. This approach is not exclusive to CAD design; it applies to every problem solving task that humans undertake, as they generally follow an iterative process. This was discussed in more detail in the paper (Simon, 1962). Although language models have become extremely good at modeling generation tasks, most of the existing models produce output in one step, which is contrasting when compared to the approach taken by humans to perform a particular task. This has become evident in many papers; some notable papers include (Vaswani et al., 2017) and (Baevski and Auli, 2018). Some papers did explore the area of editing the generation post-inferencing (Omelianchuk et al., 2020), but these papers have done the editing step only once and have not implemented refining the generation in a progressive multistep manner. Some papers like (Madaan et al., 2021) created a model of the

problem scenario before trying to attempt the problem. The drawback of this model is that it is expensive. (Reid and Neubig, 2022) proposed an architecture that takes an input sequence, passes it to an encoder with an autoregressive tag predictor, and then conditions the generation based on the predicted edit operations. This refinement model relies heavily on domain-specific data. Another direction explored to provide feedback for the generation of the model is to use reinforcement learning from human feedback (Stiennon et al., 2020) and (Ouyang et al., 2022). This work showed that it is possible to increase the quality of generation by training a model to optimize for human preferences. Although adding a human feedback layer is extremely helpful, it is very expensive and requires a lot of training data. A recent paper (Madaan et al., 2024), worked on an effective refinement approach without any external supervision. The structure outlined in this paper involves employing a language model to create an initial output that is then fed back into the same model to obtain feedback. The authors used this feedback to improve the initial output. This study has improved the generation process by approximately 20% for general tasks.

Almost all the research on feedback loops until now focused on a single generic feedback, but this approach failed to address the diverse error types found in LM-generated reasoning chains (Nathani et al., 2023). (Schick et al., 2024) introduced Toolformer, wherein an LLM learns to use external tools to solve specific problems. This is particularly useful in solving certain domain-specific problems. However, in our case, there is no such tool available as our problem is a lot more complex.

(Parakh et al., 2023) explored a new kind of human feedback (other than RLHF, which requires a lot of training data and resources). They gave feedback in between the generation steps of the LLM, and this greatly helped in the quality of the output as the LLM knew exactly what was going wrong. Even though this approach makes the system much less autonomous, it is a promising direction to explore. Considering all the approaches taken about the source of the feedback, in our case, self-feedback is the best.

Another important aspect to cover is when to correct the model with feedback. There are three options viz., training-time correction (Huang et al., 2022), generation-time correction (Yang et al., 2022), and post-hoc correction. (Pan et al., 2023) does an excellent survey on this topic. For our case, post-hoc correction is optimal as it facilitates the incorporation of more informative natural language feedback, which in turn enhances explainability.

After performing a comprehensive survey of both code generation and feedback loops, we see that strong LLMs (such as GPT-4) are able to perform code generation and are also capable of reasoning to figure out the differences between, say, two sentences. Keeping these advances in mind, we propose to generate 3D CAD models using natural language prompts using a novel feedback loop (explained in detail in Section 5). We are re-implementing the self-refine (Madaan et al., 2024) paper, particularly the code optimization task.

3 Reproduction of Self-Refine

In our reimplementation of the SELF-REFINE research paper for the code optimization part, we focused on leveraging the iterative refinement capability of large language models (LLMs), specifically Chatgpt (gpt-3.5-turbo), to enhance code generation. The SELF-REFINE method demonstrates a novel approach to refining the outputs of LLMs through iterative feedback and refinement without requiring additional supervised data, training, or reinforcement learning. In our study, we reimplement and evaluate the SELF-REFINE process, leveraging the introspective feedback mechanism to optimize functionally correct Python code generation.

3.1 Dataset and Initial Code Setup

Following the SELF-REFINE research paper, our work also utilized the CodeNet dataset. The CodeNet dataset, as introduced by (Puri et al., 2021) in 2021, is a large-scale, diverse, and high-quality dataset of code samples curated by IBM to advance the research in AI-powered code understanding and generation. It contains approximately 14 million code samples and about 500 million lines of code in more than 50 programming languages, which are sourced from online coding platforms. The fact that CodeNet comprises a diverse collection of competitive programming solutions and presents inherently complex multi-line code snippets that often challenge readability and efficiency makes CodeNet suitable for our code performance optimization task.

3.2 Introspective Feedback Generation

In the first phase of the SELF-REFINE process, the LLM examined the initial code submission and

Figure 2: Initial code, feedback given, and final code for further error analysis

identified potential areas for optimization. Drawing on its ability to understand and reason about the code, the model generated natural language feedback detailing algorithmic inefficiencies and suggesting areas for improvement.

3.3 Code Refinement through SELF-REFINE

Using the feedback provided by ChatGPT, we refined the code to generate a new, optimized version. This revised code was designed to maintain the initial code's functional correctness while improving upon its efficiency. We recorded the refined code's execution time and accuracy, which served as indicators of the optimization success.

3.4 Evaluation Metrics

We calculated several key performance metrics to quantify the optimization results: 1. % Optimized: The generated code is considered as optimized if the accuracy is 1 and the code is faster wrt the input code. Also the runtimes of the generated code must be statistically significant from that of the input code runtimes. 2. Speedup: Ratio between input code execution time and generated code execution time.

3.5 Results comparison

The original paper ran the code optimization task using 3 closed models viz., gpt3.5, Chatgpt, and gpt4. Since they are all extremely expensive models, we decided to run it only with Chatgpt. Our initial plan was to run the system on the entire CodeNet-1000 dataset, but we underestimated the cost of the self-refine system. With our budget constraints, we were able to run the system for 281 samples. The original paper did not divulge if they used the entire dataset or not, nor have they uploaded the results file in their codebase for us to infer. They have uploaded gpt4 outputs in their codebase, and from that, we see that they ran it for 573 samples only. These discrepancies prohibit us from EXACTLY comparing our results with the one reported in the paper. However, we observe the same trends that were seen in the original paper. Same as the original paper, we ran self-refinement for 4 attempts.

Table 1 shows the % code optimized for both the original paper and our runs. We can see the same

	Base (%)	**+Self-Refine (%)**
Original Paper	23.9	27.5
Our results	21.7	37.01

Table 1: % code optimized using Chatgpt as base LLM

	y0	**y1**	**y2**	**y3**
Original	22.0	27.0	27.9	28.8
Ours	21.7	32.38	35.23	37.01

Table 2: Iteration-wise score improvements (% optimized)

trend that was observed (self-refine is improving the performance). Table 2 shows an excellent case of our runs following the same trends as the original. y0 is the initial output, and y3 is the output after three refinement iterations. We can clearly see that the maximum improvement in performance comes after the first refinement (in both the original and our runs). We see diminishing returns in the improvement as the number of iterations increases. It is to be noted that the results in Table 2 of the original paper are the average of all three models that they used. There are various reasons as to why our results do not match exactly with the original paper.

1. **Dataset discrepancy:** As specified earlier, due to budget constraints, we had to cut back on the number of samples in our test data. Also, it is not clear how many samples the original paper used to get the results.

2. **Model improvements:** The original paper was published a year ago, and the code development would have been done much before that. But, OpenAI models such as ChatGPT have improved significantly from that time.

Due to these reasons, we do not exactly get the same results, but the trend is followed perfectly.

3.6 Analysis

After four iterations of the SELF-REFINE process, aimed at optimizing the code via direct optimization with ChatGPT (attempt 0) and subsequent iterations (attempts 1-3) refined based on feedback, a manual review was conducted to assess the impact of feedback on optimization performance. Initially, we targeted three case scenarios: (1) successful optimization (accuracy on test case is 1.0), (2) failed optimization due to incorrect feedback, and (3) failed optimization despite correct feedback. However, our findings revealed no instances of logically incorrect feedback. Failures generally stemmed from feedback that did not identify areas for improving code efficiency, lacking in-depth guidance on addressing edge cases, or ensuring consistency in variable usage throughout the program. Consequently, in Figure 2, we present cases of both success and failure in code optimization based on correct but missing guidance feedback on the first and last SELF-REFINE attempts.

Figure 2 demonstrates how precise feedback significantly improves code optimization by addressing both algorithmic efficiency and best coding practices. In the first sample case, the feedback advised against using Numpy arrays for tasks not suited to vectorized operations, suggested Python lists for their efficiency, and recommended replacing $eval()$ with $int()$ to enhance security and performance. This led to a notable runtime reduction, achieving a speed increase of over six times (from 1.674 measured run time to 0.257). In the third sample case, feedback highlighted the inefficiency of redundant computations, recommending a running sum to avoid repeated recalculations. These two instances underscore the effectiveness of targeted, actionable feedback in optimizing code performance.

The second and fourth examples shown in Figure 2 highlight areas for improvement in the feedback mechanism. In the second sample case shown in Figure 2, both slow code and optimized code are attempting to solve a problem involving counting the number of valid triplets (a_i, b_j, c_k) across three lists a, b, c, where each a_i is from a, b_j is from b, c_k is from c, with the condition that each $a_i < b_j < c_k$. The feedback appears to be correct in that it identifies the inefficiency of performing binary searches within a loop, leading to a quadratic time complexity $O(n^2)$. By pre-computing the counts, the time complexity can be improved. However, feedback might lack guidance on maintaining the accuracy of the computation, which is crucial for the optimized code to be considered successful. For example, assumption about the same length of a, b, and c may cause an issue: the code uses n (the length of b) to calculate c_count which may lead to incorrect results if c is not the same length as b. Given a test case, b = [5], c = [1,2,3] (in this case, all elements are less than the single element in b). For b[0] = 5, we expect c_count to be 0 because no elements in c greater than 5. But $bisect\_right(c, val)$ will return 3, $n - bisect\_right(c, val)$ will return $1 - 3$, which is -2, indicating an incorrect count. It is more logical for n to be the length of c in the context of the code. To improve feedback in the future, feedback could provide the following guidance: 1. Make sure each variable is used consistently and correctly throughout the code: in our case, if n is intended to be the size of list b, then it should not be used in calculations pertaining the lengths of a or c. 2. Suggest tests for edge cases: recommend some test cases to consider during the process of code optimization.

In the fourth sample case, the task involves calculating the minimum of two numbers, a and b, such that N = a x b, where N is given. The solution is ans = (a + b) - 2, which is a typical problem in

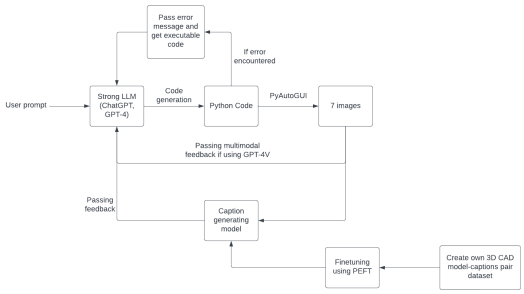

Figure 3: Proposed architecture

competitive programming that deals with finding the divisors of a number. The feedback is correct in finding NumPy's array creation is not needed for this problem and adds unnecessary overhead. The feedback is good as it targets the major inefficiency. However, it overlooks the inefficiency and potential error caused by appending two lists to determine the minimum sum and handling error in edge case N = 1, where the list div would be empty, and [N // d for d in div] would also be empty, leading to a failure. The feedback could be improved by suggesting a check for edge cases such as N = 1 and emphasizing Python-specific optimization, such as avoiding list concatenation which could lead to a time-consuming operation if div is large.

4 Proposing our project

Based on our comprehensive survey of code generation and feedback loops, we identify that there is no research done to generate 3D CAD models using natural language instructions. We propose a novel architecture consisting of a new type of feedback loop to solve this problem. The proposed architecture is shown in Figure 3. We do not have any neural baselines for this task and we are trying to set the benchmark. The decisions taken at every block in the architecture were based on our findings after doing the survey and reproduction of self-refine.

We initially pass the user prompt to a strong LLM such as ChatGPT or GPT-4. The LLM then generates a Python code to generate the 3D model in the FreeCAD (git, b) software. We then plan to use PyAutoGUI (git, a), which is a library that allows full control of the user's keyboard and mouse to run the code. If any errors are generated when running the code, we plan to pass the code and the error logs back to the LLM for it to correct the code. After we get an executable code, we take 7 images of the generated 3D model (7 different views) and then pass them to a caption-generating model such as (Li et al., 2022a). This generated caption is passed back to the LLM as feedback and is compared with the user prompt. The LLM then knows what is required (user prompt), and what it generated (feedback). Using this information, it rewrites the code. We also plan to finetune the caption-generating model to better identify CAD images. We plan to pick 200-250 images from the ABC dataset (Koch et al., 2019) and annotate them manually with natural language captions. The core idea of our feedback loop is from the above reproduced self-refine paper, but the way that we are giving feedback is novel. After doing the survey, reproduction, and error analysis, the difficulties that we predict in our project are:

1. **Requirement of strong models:** The self-refine paper tried running the same system using Vicuna-13B as their base model, but it produced extremely bad results, showing that this kind of system works well only with strong models. In our architecture, we are using the same LLM to do code generation and reasoning which means to get good results, we need strong models. But, we do not have enough budget to use ChatGPT during development. We can use strong open-source models during development, but optimizing the prompt for the closed models will be difficult with limited runs in our hands.
2. **Handling edge cases:** Our error analysis showed that self-refine is not good at handling edge cases. This shows that we have to structure our prompts in a much better way rather than naively asking the difference between the user query and the generated caption.

Our final goal is to set a SOTA for 3D CAD generation and provide a benchmark dataset to propel further work in this area.

5 Conclusion

A detailed survey was conducted on language models used for code generation and the methods used to provide feedback to the model for correcting its previous generations. Based on the literature survey we performed, the paper "SELF-REFINE: Iterative Refinement with Self-Feedback" resembles the refinement loop in our proposed architecture. The results generated by the model and their corresponding analysis were discussed in detail and the architecture of our project was proposed along with the challenges we predict during development.

References

a. GitHub - asweigart/pyautogui: A cross-platform GUI automation Python module for human beings. Used to programmatically control the mouse & keyboard. — github.com. https://github.com/asweigart/pyautogui.git. [Accessed 31-03-2024].

b. GitHub - FreeCAD/FreeCAD: This is the official source code of FreeCAD, a free and opensource multiplatform 3D parametric modeler. — github.com. https://github.com/FreeCAD/FreeCAD.git. [Accessed 31-03-2024].

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Alexei Baevski and Michael Auli. 2018. Adaptive input representations for neural language modeling. *arXiv preprint arXiv:1809.10853*.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2022. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Wenpin Hou and Zhicheng Ji. 2024. A systematic evaluation of large language models for generating programming code. *arXiv preprint arXiv:2403.00894*.

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2022. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*.

Shashank Mohan Jain. 2022. Hugging face. In *Introduction to transformers for NLP: With the hugging face library and models to solve problems*, pages 51–67. Springer.

Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. 2019. Abc: A big cad model dataset for geometric deep learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9601–9611.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

Junnan Li, Dongxu Li, Caiming Xiong, and Steven C. H. Hoi. 2022a. BLIP: bootstrapping language-image pre-training for unified vision-language understanding and generation. *CoRR*, abs/2201.12086.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023a. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, and Shing-Chi Cheung. 2023b. Finding failure-inducing test cases with chatgpt. *arXiv preprint arXiv:2304.11686*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022b. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Bingchang Liu, Chaoyu Chen, Cong Liao, Zi Gong, Huan Wang, Zhichao Lei, Ming Liang, Dajun Chen, Min Shen, Hailian Zhou, et al. 2023. Mftcoder: Boosting code llms with multitask fine-tuning. *arXiv preprint arXiv:2311.02303*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Aman Madaan, Niket Tandon, Dheeraj Rajagopal, Peter Clark, Yiming Yang, and Eduard Hovy. 2021. Think about it! improving defeasible reasoning by first modeling the question scenario. *arXiv preprint arXiv:2110.12349*.

Deepak Nathani, David Wang, Liangming Pan, and William Yang Wang. 2023. Maf: Multi-aspect feedback for improving reasoning in large language models. *arXiv preprint arXiv:2310.12426*.

Kostiantyn Omelianchuk, Vitaliy Atrasevych, Artem Chernodub, and Oleksandr Skurzhanskyi. 2020. Gector–grammatical error correction: tag, not rewrite. *arXiv preprint arXiv:2005.12592*.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback, 2022. *URL https://arxiv.org/abs/2203.02155*, 13:1.

Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. 2023. Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies. *arXiv preprint arXiv:2308.03188*.

Meenal Parakh, Alisha Fong, Anthony Simeonov, Tao Chen, Abhishek Gupta, and Pulkit Agrawal. 2023. Lifelong robot learning with human assisted language planners. In *CoRL 2023 Workshop on Learning Effective Abstractions for Planning (LEAP)*.

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training.

Partha Pratim Ray. 2023. Chatgpt: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope. *Internet of Things and Cyber-Physical Systems*.

Machel Reid and Graham Neubig. 2022. Learning to model editing processes. *arXiv preprint arXiv:2205.12374*.

Aditi Roy, Ioannis Akrotirianakis, Amar V Kannan, Dmitriy Fradkin, Arquimedes Canedo, Kaushik Koneripalli, and Tugba Kulahcioglu. 2020. Diag2graph: Representing deep learning diagrams in research papers as knowledge graphs. In *2020 IEEE International Conference on Image Processing (ICIP)*, pages 2581–2585. IEEE.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36.

Herbert A Simon. 1962. The architecture of complexity. *Proceedings of the American philosophical society*, 106(6):467–482.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. 2020. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Kaiyu Yang, Jia Deng, and Danqi Chen. 2022. Generating natural language proofs with verifier-guided search. *arXiv preprint arXiv:2205.12443*.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*, pages 476–486.