

Self-collaboration Code Generation via ChatGPT

Yihong Dong*, **Xue Jiang***, **Zhi Jin[†]**, **Ge Li[†]**

Key Laboratory of High Confidence Software Technologies (Peking University),
Ministry of Education; Institute of Software, Peking University, Beijing, China
 {dongyh, jiangxue}@stu.pku.edu.cn, {zhijin, lige}@pku.edu.cn

Abstract

Although Large Language Models (LLMs) have demonstrated remarkable code-generation ability, they still struggle with complex tasks. In real-world software development, humans usually tackle complex tasks through collaborative teamwork, a strategy that significantly controls development complexity and enhances software quality. Inspired by this, we present a self-collaboration framework for code generation employing LLMs, exemplified by ChatGPT. Specifically, through role instructions, 1) Multiple LLMs act as distinct “experts”, each responsible for a specific subtask within a complex task; 2) Specify the way to collaborate and interact, so that different roles form a virtual team to facilitate each other’s work, ultimately the virtual team addresses code generation tasks collaboratively without the need for human intervention. To effectively organize and manage this virtual team, we incorporate software-development methodology into the framework. Thus, we assemble an elementary team consisting of three ChatGPT roles (i.e., analyst, coder, and tester) responsible for software development’s analysis, coding, and testing stages. We conduct comprehensive experiments on various code-generation benchmarks. Experimental results indicate that self-collaboration code generation relatively improves 29.9%-47.1% Pass@1 compared to direct code generation, achieving state-of-the-art performance and even surpassing GPT-4. Moreover, we showcase that self-collaboration could potentially enable LLMs to efficiently handle complex real-world tasks that are not readily solved by direct code generation, as evidenced in case study.

1 Introduction

Code generation aims to generate code that satisfies human requirements expressed in the form of some specification. Successful code generation can improve the efficiency and quality of software development, even causing changes in social production modes. Therefore, code generation has been a significant research hotspot in the fields of artificial intelligence, natural language processing, and software engineering. Recently, code generation has made substantial advancements in both academic and industrial domains [Chen et al., 2021, Shen et al., 2022, Li et al., 2022, Dong et al., 2023a]. In particular, LLMs have achieved excellent performance and demonstrate promising potential on code generation tasks [Nijkamp et al., 2022, Fried et al., 2022, Zheng et al., 2023].

Nonetheless, generating correct code for complex requirements poses a substantial challenge, even for experienced human programmers. Intuitively, humans, as social beings, tend to rely on collaborative teamwork when encountering complex tasks. Teamwork through division of labor, interaction, and collaboration to solve complex problems, which has been theorized to play an important role in dealing with complexity, as posited in both teamwork theory [Belbin, 2012, Katzenbach and Smith, 2015] and software engineering practice [Beck et al., 2001, McChesney and Gallagher, 2004,

*Equal Contribution

[†]Corresponding author

DeMarco and Lister, 2013]. The benefits of collaborative teamwork are manifold: 1) It breaks down complex tasks into smaller subtasks, making the entire code generation process more efficient and controllable. 2) It assists with error detection and quality control. Team members can review and test the generated code, providing feedback and suggestions for improvement, thus reducing potential errors and defects. 3) It ensures that the generated code is consistent with the expected requirements. Team members can offer different viewpoints to solve problems and reduce misunderstandings.

A straightforward way to implement collaborative teamwork entails training different models to handle the corresponding subtasks, subsequently conducting joint training to foster mutual understanding of behaviors to assemble them into a team [Schick et al., 2022]. However, this training approach is costly, especially for LLMs. The scarcity of relevant training data further exacerbates the difficulty of achieving collaborative code generation. Revolutionary advancements in artificial general intelligence (AGI), especially LLMs represented by ChatGPT [OpenAI], provide a turning point. These LLMs perform commendably across tasks in various stages of software development, laying the groundwork for division of labor. Furthermore, LLMs use language as the foundation for input and output and align with human needs through instructions or prompts, offering the potential for inter-model interaction and collaboration.

To this end, we propose a self-collaboration framework aimed at guiding LLMs to collaborate with themselves, thereby dealing with complex requirements and boosting the performance of code generation. This framework is divided into two parts: division of labor and collaboration, both of which are dominated by role instructions. First, role instructions achieve division of labor by allocating specific roles and responsibilities to LLMs. This strategy enables LLMs to think about and tackle tasks from the standpoint of the roles they play, transforming the original LLMs into domain “experts”, as shown in Fig. 1. Second, role instructions control the interaction between roles, allowing otherwise isolated roles to form a virtual team and facilitate each other’s work.

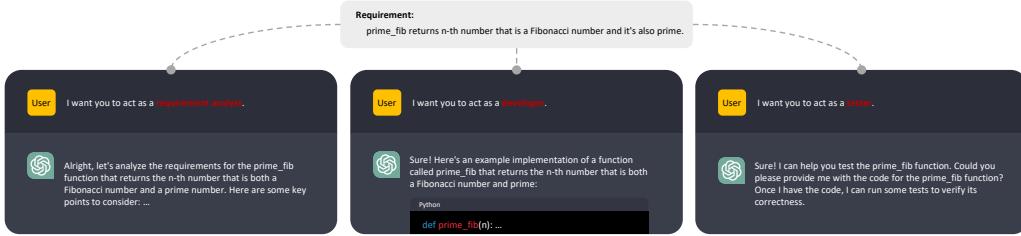


Figure 1: An example of role-playing. Through role-playing, LLM transforms into an expert within a specific domain, delivering a professional-perspective response to the same requirement.

To promote efficient collaboration, we introduce software-development methodology (SDM) into self-collaboration framework. SDM provides a set of clearly defined stages, principles, and practices that serves to organize and manage teams effectively, ultimately controlling development complexity and improving software quality [Abrahamsson et al., 2002, Ruparelia, 2010]. Following SDM, we instantiate an elementary team composed of three roles (i.e., analyst, coder, and tester). These roles adhere to a workflow where the stages of analysis, coding, and testing are performed sequentially, with each stage providing feedback to its predecessor. Specifically, the analyst breaks down requirements and develops high-level plans for guiding the coder. The coder then creates or improves code based on the plans or the tester’s feedback. Concurrently, the tester compiles test reports based on the coder’s outcome and documents any issues found during testing. We employ three ChatGPT agents to respectively play the three roles through role instructions, and then they collaborate to address code generation tasks under the guidance of self-collaboration framework.

The primary contributions of our work can be summarized as follows: (1) We propose a self-collaboration framework with role instruction, which allows LLMs to collaborate with each other to generate code for complex requirements. (2) Following SDM, we instantiate an elementary team, which comprises merely three ChatGPT roles (i.e., analyst, coder, and tester) responsible for their respective stages in the software development process. (3) Building on self-collaboration framework, the virtual team formed by ChatGPT (GPT-3.5) can achieve state-of-the-art performance on multiple code-generation benchmarks, even surpassing GPT-4. (4) In some selected real-world scenarios, self-collaboration code generation exhibits notable effectiveness on complex code generation tasks that are challenging for direct code generation.

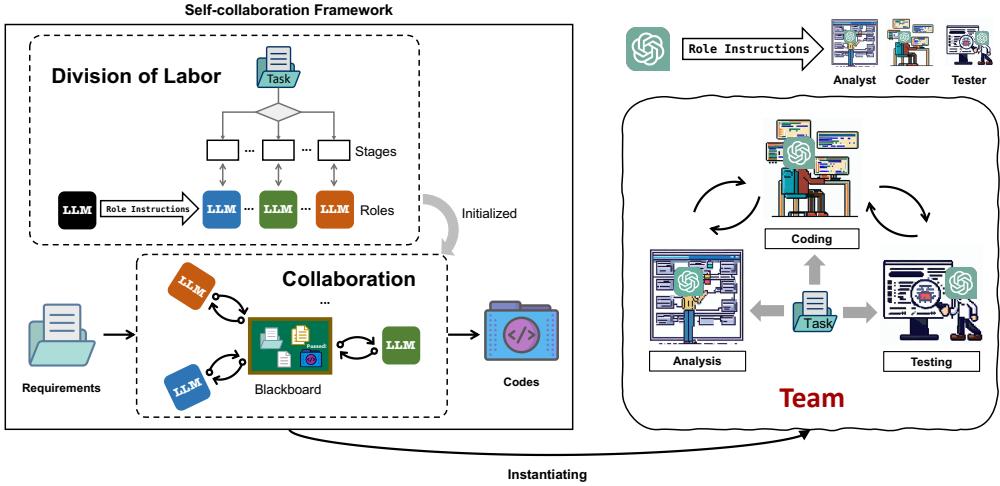


Figure 2: Self-collaboration framework for code generation and its instance.

2 Self-collaboration Framework

Our self-collaboration framework consists of two parts: division of labor (DOL) and collaboration, which is shown in Fig. 2 (left). Given a requirement x , we propose to perform self-collaboration with LLMs to generate the output y . The task is defined as $\mathcal{T} : x \rightarrow y$.

2.1 Division of Labor

In DOL part, we leverage prior knowledge to decompose the process of solving a complex task into a series of stages $\mathcal{T} \Rightarrow \{\mathcal{S}_i\}_{i=1}^l$ and construct some distinct roles $\{R_j\}_{j=1}^m$ based on $\{\mathcal{S}_i\}_{i=1}^l$. Each stage \mathcal{S}_i can be processed by one or more roles R'_j s.

It is widely acknowledged that LLMs are sensitive to context, as they are trained to predict subsequent tokens based on preceding ones. Consequently, it is prevalent to control LLM generation using instructions or prompts [Ouyang et al., 2022, Chung et al., 2022, OpenAI, 2023]. In order to achieve division of labor, we craft a specific type of instruction to assign roles and responsibilities to LLMs, which we refer to as role instructions. Specifically, we ask an LLM to act as a particular role that has a strong correlation with its responsibilities. Furthermore, we need to convey the detailed tasks, i.e. responsibilities, this role should perform. In general, the more detailed the task is described in the instruction, the better the LLM will behave in line with your expectations. One case where it may not be necessary to outline a role's responsibilities is when the division of labor is common enough that matching roles can be found in reality.

Through role-playing, we can effectively situate LLM within a specific domain, thereby eliciting its expertise within that domain. Our empirical evidence suggests that this role-playing approach yields superior results compared to directly engaging the LLM in the task without a pre-defined contextual setting. Thus, role-playing can be harnessed as an efficient tool to enhance the performance of the LLM in specialized tasks.

Note that role instructions only need to be provided once at the initialization of each LLM agent. In subsequent interactions, the roles will automatically interact with each other and pass information without the need to enter further instructions. This enhances the overall efficiency and clarity of subsequent interaction and collaboration.

2.2 Collaboration

After assigning roles to LLMs in DOL part, roles interact their outputs with other roles as the stages progress, refining the work and ensuring an accurate and thoughtful output y . In collaboration part, we focus on facilitating effective interactions among distinct roles to ensure that they mutually enhance each other's work.

The interaction among roles occurs in the form of natural language (NL), which is supported by the foundational aspects of the language model. We specify the roles, information, and format of each role interaction in role instructions, which allows the whole process of collaboration to be well controlled. The collaboration part can be formalized as follows:

$$\arg \max_{s_t} P(s_t | s_{\{<t\}}, R_{m(S_t)}, x), \quad (1)$$

where s_t is the output of stage S_t , $s_{\{<t\}}$ indicates the prerequisite-stage outputs of S_t ³, and $R_{m(S_t)}$ represents the role corresponding to S_t . We consider the computation of $P(s_t | s_{\{<t\}}, R_{m(S_t)}, x)$ as the collaboration, wherein role $R_{m(S_t)}$ collaborates with the roles of each preceding stage to generate s_t . Output y is iteratively updated along with the progression of S_t :

$$y_t = f(s_t, y_{<t}) \quad (2)$$

where f is an update function. Once the end condition is satisfied, the final output y is derived. To coordinate collaboration between different roles, we set up a shared blackboard [Nii, 1986], from which each role exchanges the required information to accomplish their respective tasks s_t via Eq. (1). The pseudocode of self-collaboration framework is outlined in Algorithm 1.

3 Instance

We introduce the classic waterfall model [Petersen et al., 2009] from SDM into self-collaboration framework to make the teamwork for code generation more efficient. Specifically, we design a simplified waterfall model consisting of three stages, i.e. analysis, coding, and testing, as an instance for self-collaboration code generation. The workflow of this instance follows the waterfall model flowing from one stage to the next, and if issues are found, it returns to the previous stage to refine. Thus, we establish an elementary team, comprising an analyst, coder, and tester, responsible for the analysis, coding, and testing stages, as illustrated in Fig. 2 (right). These three different roles are assigned the following tasks:

Analyst. The goal of an analyst is to develop a high-level plan and focus on guiding the coder in writing programs, rather than delving into implementation details. Given a requirement x , the analyst decomposes x into several easy-to-solve subtasks that facilitate the division of functional units and develops a high-level plan that outlines the major steps of the implementation.

Coder. As the central role of this team, the coder receives plans from an analyst or test reports from a tester throughout the workflow. Thus, we assign two responsibilities to the coder: 1) Write code that fulfills the specified requirements, adhering to the plan provided by the analyst. 2) Repair or refine code, taking into account the feedback of test reports feedbacked by the tester.

Tester. The tester acquires the code authored by the coder and subsequently documents a test report containing various aspects, such as functionality, readability, and maintainability. Rather than generating test cases and then manually testing the code by execution, we advocate for a practice where the model simulates the testing process and produces test reports, thereby facilitating interaction and avoiding external efforts.

We customize role instructions for LLMs (exemplified by ChatGPT) to play the three roles. An example of role instruction for coder is shown in Fig. 3. In this example, the role instruction includes not only the role description (role and its responsibilities), but also the team description and user requirements, which will work together to initialize the ChatGPT agent, thereby setting the behavior of ChatGPT. In addition, interactions occur between roles responsible for two successive stages, and we limit the maximum interaction to n . We update the output y_t only when the stage S_t is coding, and this workflow terminates upon n is reached or the tester confirms that no issues persist with y_t .

³Note that our self-collaboration framework can be parallelized if the relationship between stages $\{S_i\}_{i=1}^l$ is not a straightforward linear relationship.

Role Instructions = Team Description + User Requirement + Role Description		
Team Description	<p>There is a development team that includes a requirements analyst, a developer, and a quality assurance tester. The team needs to develop programs that satisfy the requirements of the users. The different roles have different divisions of labor and need to cooperate with each others.</p>	
User Requirement	<p>The requirement from users is '{Requirement}'.</p> <p><i>For example: {Requirement} = Input to this function is a string containing multiple groups of nested parentheses. Your goal is to separate those group into separate strings and return the list of those. Separate groups are balanced (each open brace is properly closed) and not nested within each other. Ignore any spaces in the input string</i></p>	
Role Description	<p>Coder:</p> <p>I want you to act as a developer on our development team. You will receive plans from a requirements analyst or test reports from a tester. Your job is split into two parts:</p> <ol style="list-style-type: none"> If you receive a plan from a requirements analyst, write code in Python that meets the requirements following the plan. Ensure that the code you write is efficient, readable, and follows best practices. If you receive a test report from a tester, fix or improve the code based on the content of the report. Ensure that any changes made to the code do not introduce new bugs or negatively impact the performance of the code. <p>Remember, do not need to explain the code you wrote.</p>	

Figure 3: An example of role instruction for coder in the instance of our self-collaboration framework.

4 Experiment Setup

Benchmarks. We perform a comprehensive evaluation on four code-generation benchmarks to demonstrate the efficacy of our self-collaboration approach. **MBPP** (sanitized version) [Austin et al., 2021] contains 427 manually verified Python programming tasks, covering programming fundamentals, standard library functionality, and more. Each task consists of an NL description, a code solution, and 3 automated test cases. **HumanEval** [Chen et al., 2021] consists of 164 handwritten programming tasks, proposed by OpenAI. Each task includes a function signature, NL description, function body, and several unit tests (average 7.7 per task). **MBPP-ET** and **HumanEval-ET** [Dong et al., 2023b] are expanded versions of MBPP and HumanEval with over 100 additional test cases per task. This updated version includes edge test cases that enhance the reliability of code evaluation compared to the original benchmark.

Settings & Baselines. In this paper, we employ two prevalent settings for code generation: (1) **NL + signature + public test cases**. This setting is set for comparison with other code generation approaches and it provides an NL description, function signature, and public test cases as input prompts. In this setting, We compare our self-collaboration approach with LLMs customized for code (AlphaCode [Li et al., 2022], Incoder [Fried et al., 2022], CodeGeeX [Zheng et al., 2023], CodeGen [Nijkamp et al., 2022], and CodeX [Chen et al., 2021]), previous state-of-the-art approach (CodeX + CodeT [Chen et al., 2022]), and generalist LLMs (ChatGPT [OpenAI], and GPT-4 [OpenAI, 2023]). Due to commercial constraints, some of the compared models are not available. Therefore, we reference the results reported in their original papers. (2) **NL-only**. This setting exclusively utilizes the NL description as an input prompt, which is more consistent with real-world development scenarios. In this setting, we further explore some complex code generation tasks in addition to the preceding benchmarks. Detailed discussions of settings can be found in Appendix B.

In all experiments, we invoke ChatGPT (GPT-3.5) through its API, namely gpt-3.5-turbo. We employ ‘0301’ version of gpt-3.5-turbo as our base model to minimize the risk of unexpected model changes affecting the results. To increase the stability of LLM’s output, we set the decoding temperature to 0. Unless otherwise stated, we use NL-only setting for code generation, and the maximum number of interactions between LLMs is limited to 4.

Metric. We adopt the unbiased variant of Pass@k [Chen et al., 2021] to measure the functional correctness of top-k generated codes by executing test cases, which can be formulated as:

$$\text{Pass}@k = \mathbb{E}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]. \quad (3)$$

In this paper, we are mainly concerned with **Pass@1**, which is a representative of the Pass@k family, because in real-world scenarios we usually consider only single generated code.

5 Experimental Results

Our experimental results include both quantitative and qualitative evaluations for code generation. Quantitative evaluations provide analysis and objective comparisons of extensive data. Qualitative evaluations, presented as case studies, provide in-depth insight and interpretation of approaches.

5.1 Main Results

The main results of self-collaboration approach compared to other approaches are shown in Table 1. The experimental results reveal that our self-collaboration framework significantly enhances the performance of the base LLMs. Remarkably, even with a simple three-member team (including an analyst, coder, and tester), self-collaboration code generation based on ChatGPT (GPT-3.5) achieves the best performance across four code-generation benchmarks, surpassing even GPT-4. When compared to ChatGPT (GPT-3.5), the improvement offered by our framework is substantial, with a relative increase ranging from 29.9% to 34.6%. It is noteworthy that the self-collaboration code generation yields more significant improvements on the datasets associated with extended test cases, namely HumanEval-ET and MBPP-ET. This suggests that our framework can effectively assist base LLMs in generating reliable code. This enhancement may be attributed to the collaborative team’s ability to consider a wider range of boundary conditions and address common bugs.

Table 1: Comparison of self-collaboration and other approaches (with their setting, i.e., NL + signature + public test cases).

Approach	HumanEval	HumanEval-ET	MBPP	MBPP-ET
AlphaCode (1.1B)	17.1	-	-	-
InCoder (6.7B)	15.2	11.6	17.6	14.3
CodeGeeX (13B)	18.9	15.2	26.9	20.4
CodeGen-Mono (16.1B)	32.9	25.0	38.6	31.6
Codex (175B)	47.0	31.7	58.1	38.8
+ CodetT	65.8	51.7	67.7	45.1
PaLM Codex (540B)	36.0	-	47.0	-
GPT-4	67.0	-	-	-
ChatGPT (GPT-3.5)	57.3	42.7	52.2	36.8
+ Self-collaboration	74.4 (\uparrow 29.9%)	56.1 (\uparrow 31.4%)	68.2 (\uparrow 30.7%)	49.5 (\uparrow 34.6%)

5.2 The Effect of Role-playing

We investigated code generation relying solely on NL descriptions. In this setting, we evaluate the performance of each ChatGPT role in the elementary team instantiated by our self-collaboration framework, as illustrated in Table 2. The experimental results show that the performance substantially improves when compared to employing only the coder role after forming a team, regardless of whether it is a two-role or three-role team. The coder-analyst-tester team achieved the best results on HumanEval and HumanEval-ET benchmarks, with relative improvements of 40.8% and 47.1%, respectively. In contrast, the coder-tester team attained the highest performance on MBPP and MBPP-ET benchmarks, with relative improvements of 36.7% and 39.4%, respectively. The suboptimal performance of the analyst on MBPP and MBPP-ET benchmarks may be attributed to the relatively basic programming tasks of MBPP, which do not necessitate planning (As problems become more complex, planning is an influential part). Moreover, some tasks in MBPP feature inputs and outputs that diverge from conventional human coding practices and cognitive processes, and they lack any hints provided in the input. Those can be easily resolved by laying off the analyst on the team and providing input-output formatting tips.

Table 2: Effectiveness of each ChatGPT role in self-collaboration code generation.

Roles	HumanEval	HumanEval-ET	MBPP	MBPP-ET
Coder	45.1	35.3	47.5	34.3
+ Analyst	54.9 (\uparrow 21.8%)	45.2 (\uparrow 28.1%)	54.8 (\uparrow 15.4%)	39.9 (\uparrow 16.4%)
+ Tester	56.8 (\uparrow 26.0%)	45.2 (\uparrow 28.1%)	64.9 (\uparrow 36.7%)	47.8 (\uparrow 39.4%)
+ Analyst + Tester	63.5 (\uparrow 40.8%)	51.9 (\uparrow 47.1%)	55.5 (\uparrow 16.9%)	40.8 (\uparrow 19.0%)

To further verify the efficacy of the role-playing strategy, we conducted a comparative analysis with two baselines without role-playing: Instruction (zero-shot), which involves instructions with the role-playing component excised, and few-shot prompting, which provides examples demonstrating the tasks of LLMs at each stage. Details of instructions and prompts are provided in Appendix B.1. The results, as presented in Table 3, show that role-playing approach substantially outperforms the baselines without role-playing. We suppose that the possible reason for the better performance of role-playing LLMs is that it provides a specific context that constrains the generation space of LLMs, making it reason within the constraints of the scenario, generating responses that align with the perspectives that LLM in that role might have. Therefore, role-playing serves to evoke the latent abilities of LLMs instead of directly improving LLM’s abilities. Moreover, in two scenarios without role-playing, we observe that instruction (zero-shot) is slightly better than few-shot prompting. We identified two potential factors that could lead to this observation. First, few-shot prompting may bias the LLMs’ understanding of human intent due to the limited selection of examples might not fully reflect intent. Secondly, the long prompt (about 14 times the instruction length in the experiment) used in few-shot prompting could hinder the LLMs’ effective extraction of relevant information.

5.3 The Effect of Interaction

To evaluate the impact of interaction on self-collaboration code generation, we controlled MI in this experiment, with the results shown in Table 4. The MI value equal to zero signifies a complete absence of interaction between the roles involved. This means that the code generated by a particular coder does not receive any form of feedback from other roles, so the result is the same as employing a coder only. The MI value increases from 0 to 1 for the largest performance improvement. This result shows that for these four benchmarks, our approach solves most of the tasks within two rounds (i.e., one round of interaction). The continued increase in the MI value beyond the initial round yields diminishing returns in terms of improvement; however, there is still a consistent enhancement observed. This suggests that more complex tasks are undergoing continuous improvement. In general, higher MI values yield better outcomes. Nonetheless, due to the constraint of maximum tokens, our exploration was limited to the MI value from 1 to 4 rounds.

5.4 Case Study

We conduct case studies to qualitatively evaluate the performance of our approach. Our study mainly focuses on two kinds of cases. The first is a relatively simple requirement for function-level code generation, sourced from Humaneval benchmark. The second involves a more complex real-world requirement, specifically asking model to create a game in Python, which is a high-level requirement similar to a user story.

Case Study on Humaneval Benchmark We evaluate self-collaboration code generation on HumanEval benchmark. A case study depicting this evaluation is presented in Fig. 4, which exemplifies the workflow and performance of our approach. 1. The analyst conducts a comprehensive analysis and devises a plan to address the entire set of requirements. For this requirement, the analyst first decomposes it into several easy-to-solve subtasks and then gives some high-level plans based on the subtasks. 2. The coder implements a code according to the given requirements, the devised decompositions, and high-level plans. It is evident that each sub-function and step within this implemented code exhibits a nearly one-to-one correspondence with decompositions and high-level plans. 3. The tester composes a detailed test report for the implemented code, identifying errors within it. In this test report, the tester thinks that the implemented code may lead to the removal of duplicate elements from the list, potentially resulting in the failure of some edge test cases. Consequently, it is recommended to eliminate the line "lst = list(set(lst))" from the implemented code. 4. The coder subsequently refines the code based on the feedback provided in the test report. In the revised code,

Table 3: The effect of role-playing for self-collaboration code generation.

Approach	HumanEval	HumanEval-ET
No Role-playing		
Few-shot prompting	58.1	47.5
Instruction (zero-shot)	60.0	47.5
Role Instruction	64.4	51.9

We sample 4 HumanEval tasks to construct examples for few-shot prompting, which are excluded in baseline evaluations for fairness.

Table 4: The effect of maximum interaction (MI) for self-collaboration code generation.

MI	HumanEval	HumanEval-ET	MBPP	MBPP-ET
0	45.1	35.3	47.5	34.3
1	60.4	50.7	53.0	38.0
2	62.2	50.7	54.1	38.5
4	63.5	51.9	55.5	40.8

the recommendation from the test report is taken into account, and the line "lst = list(set(lst))" is concurrently removed. 5. The tester assessed the revised code, confirming that no issues exist and all tests pass, thus concluding the code generation process.

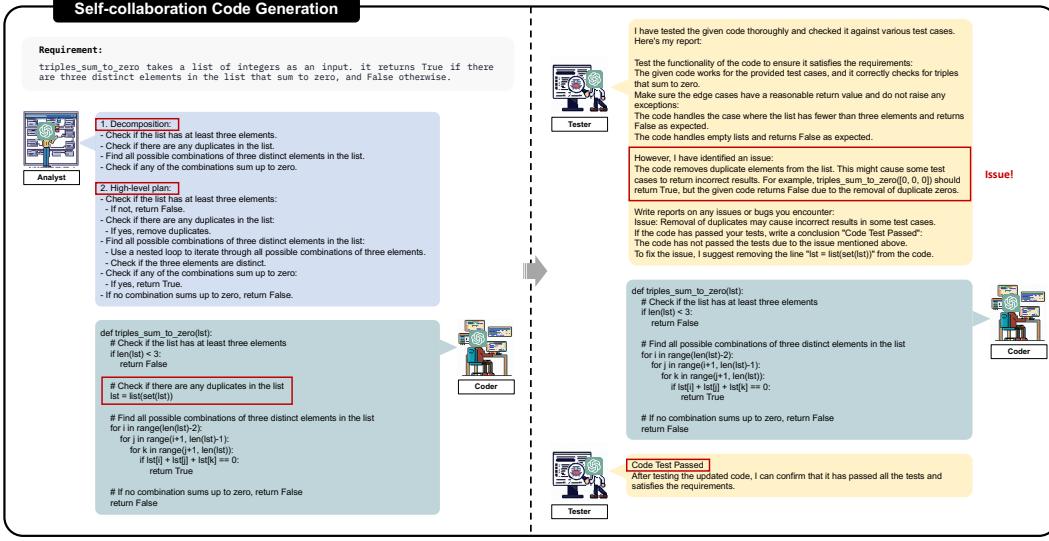


Figure 4: Case study on HumanEval benchmark. Inputs of the roles are omitted for clarity and focus on the progression of the workflow.

Case Study on Complex Task We apply self-collaboration code generation to a complex game development requirement, with a case study shown in Fig. 5. Self-collaboration approach generates a working game that fulfills all requirements without human intervention. First, our approach implements the correct game logic, where the mouse clicks to start the game, and the mouse controls the character from the starting point, through all obstacles, and avoiding falling bombs to reach the ending point. The game design ensures precise character control, feedback for win/loss conditions, and correct collision detection with obstacles&bombs. Second, the specifications outlined in the requirements have been adhered to, including the visual representation of the start and end points, the necessary game assets loading, and the proper scaling of images. Third, it also notices some game logic that is not mentioned in the requirements, but perfectly in line with common sense, such as "Bombs fall from the top of the screen and reset their position when they hit the bottom". In contrast, the result of direct generation is a rough draft of the Python script. This script does not include all the functionality requested in the requirements. Even if we manually input the instruction "Continue to add functionality" until ChatGPT thinks all functionality is covered, it still fails to satisfactorily fulfill this requirement actually. Full details of this case study are shown in Appendix C.

6 Related Work

In this section, we outline the probably most relevant directions and associated papers of this work. Extended discussions of related work can be found in Appendix D.

There are a few works that use various stages of software development to enhance code generation based on LLMs without retraining. Some existing approaches pick the correct or best solution from multiple candidates by reranking, e.g., CodeT [Chen et al., 2022] and Coder-Reviewer [Zhang et al., 2022]. They provide no feedback to model, so that falls under the category of post-processing. Recently, another type of approaches focus on improving the quality of generated code. Self-planning [Jiang et al., 2023] introduces a planning stage prior to code generation. Self-debugging [Chen et al., 2023] teaches LLM to perform rubber duck debugging after code generation. They rely on few-shot prompting to instruct the model, which requires a certain amount of effort to construct specific demonstrations as prompting examples for each dataset.

Our work proposes self-collaboration code generation, which can cover any stage of the software development process with little effort to customize the role instructions. Each role can be regarded

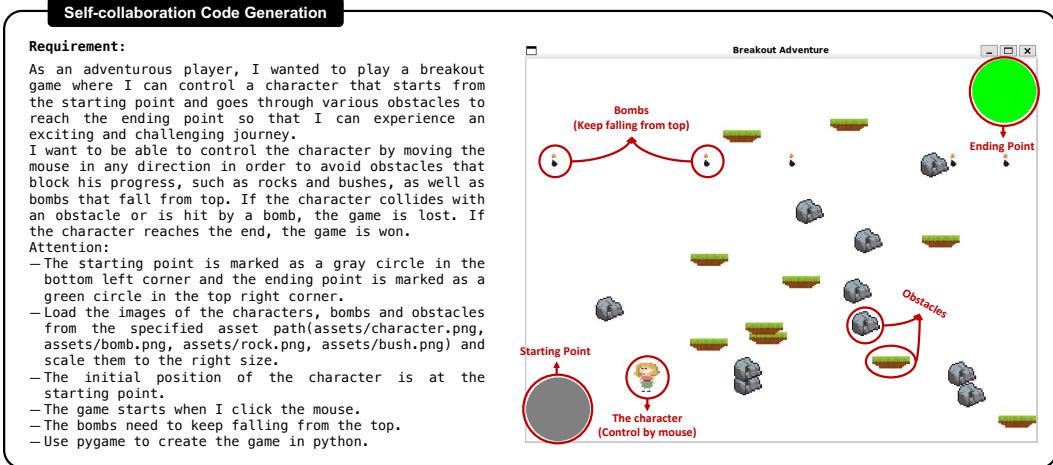


Figure 5: Case study on complex tasks in real-world scenarios. Red markers, added post-process, are employed to denote the specific objects.

as an “expert” responsible for a distinct stage and provides feedback to each other to improve the quality of final generated code. By introducing SDM, we achieve steady improvements in quantitative evaluation with the uniform virtual development team.

7 Discussion and Future Work

In this section, we discuss some limitations of the virtual development team we currently instantiate and present a set of potential directions for future research. 1) In this paper, we assemble an elementary team consisting of only three roles. However, the self-collaboration framework allows for easy modification of the team composition to accommodate different practical needs. For instance, drawing from the practice of pair programming, we can establish a two-person team for agile development, with one role responsible for code generation and the other taking on its review. Furthermore, Our approach cannot be restricted by traditional software engineering methodologies. In the AGI era, we can create new software development models and virtual software development teams composed of completely new roles. 2) The team that we assembled is a fully autonomous system, which may work away from requirements and affect the effectiveness of systems. In order to address this issue, incorporating the guidance of a human expert to oversee the operations of the virtual team may be necessary. This approach not only ensures that the virtual team operates within established requirements, but also offers several benefits: I. It reduces labor costs by eliminating a large amount of human work; II. It improves communication and development efficiency as compared to traditional software development teams, leading to a more streamlined and efficient development process. 3) Beyond working with the model itself, we can also employ external tools to compensate for the limitations intrinsic to LLM’s capabilities. Software engineering development has accumulated numerous tools so far. Toolformer [Schick et al., 2023] is proposed to provide direction on how we can guide LLMs to use these tools. It is worth exploring the incorporation of calling these tools through role instructions in self-collaboration framework.

8 Conclusion

In this paper, we propose a self-collaboration framework designed to enhance the problem-solving capability of LLMs in a collaborative and interactive way. We investigate the potential of ChatGPT in facilitating collaborative code generation within software development processes. Specifically, based on the proposed framework, we assemble an elementary team consisting of three distinct ChatGPT roles, designed to address code generation tasks collaboratively. Extensive experimental results demonstrate the effectiveness and generalizability of self-collaboration framework. We believe that enabling models to form their own teams and collaborate in accomplishing complex tasks is a crucial step toward automating software development.

Appendix A Extended Experimental Results

Self-collaboration for different chat-based LLMs We investigate the self-collaboration capacities of chat-based LLMs across varied model sizes, specifically Fastchat-3B [LMSYS, a], ChatGLM-6B [THUDM], MPT-7B [MosaicML], Vicuna-7B/13B [LMSYS, b], HuggingChat-30B [Huggingface], Dromedary-65B [IBM], and ChatGPT (GPT3.5) [OpenAI]. Our objective is to evaluate the efficacy of the self-collaboration approach in tackling complex tasks, specifically those challenging for direct code generation. For such tasks, we employ the self-collaboration strategy as a solution. As illustrated in Fig. 6, the coding ability of chat-based LLMs generally exhibits an increasing trend with the enlargement of model sizes. The self-collaboration capability starts to manifest itself around the 7B parameters and subsequently continues to escalate, serving to evoke latent intelligence within LLMs.

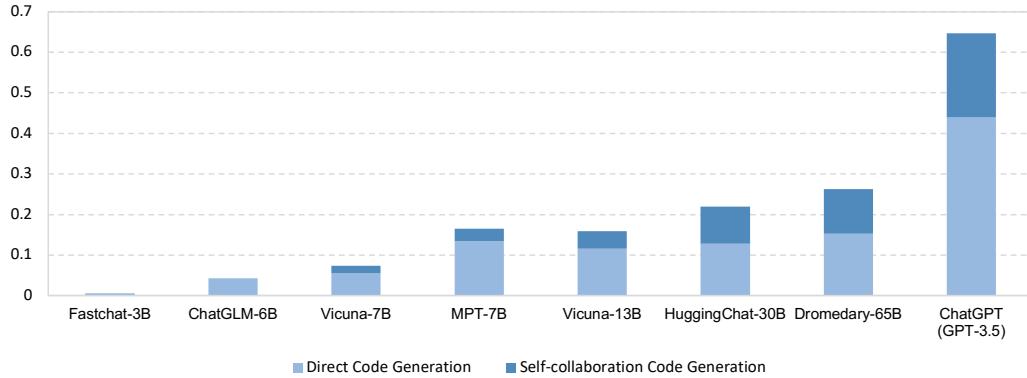


Figure 6: Self-collaboration capacities of chat-based models across varied model sizes.

Performance of self-collaboration code generation with the first setting We verify the performance of our self-collaboration code approach in the settings where signatures and public test cases are provided in addition to NL. The results of this evaluation are displayed in Table 5. As demonstrated in the findings, the self-collaboration code approach is far more effective than direct code generation, which focuses only on the coding phase. Furthermore, in this experimental setup, the role of analysts is particularly pronounced. In numerous cases, the influence of the Analyst-Coder is nearly equivalent to that of the entire team. In the absence of analysts, testers are able to detect coding errors to a certain degree. However, integrating analysts, coders, and testers into cohesive teams maximizes efficiency and yields superior results.

Table 5: Performance of self-collaboration code generation with the first setting (i.e., NL + signature + public test cases).

Approach	HumanEval	HumanEval-ET	MBPP	MBPP-ET
Direct	57.3	42.7	52.2	36.8
Self-collaboration (Virtual Team)				
Analyst-Coder	73.5 (\uparrow 28.2%)	55.2 (\uparrow 29.2%)	64.2 (\uparrow 25.1%)	48.9 (\uparrow 32.9%)
Coder-Tester	60.3 (\uparrow 5.3%)	45.2 (\uparrow 5.9%)	64.2 (\uparrow 23.0%)	48.3 (\uparrow 31.3%)
Analyst-Coder-Tester	74.4 (\uparrow 29.9%)	56.1 (\uparrow 31.4%)	68.2 (\uparrow 30.7%)	49.5 (\uparrow 34.6%)

Extended case study on MBPP / HumanEval benchmark In addition to the case study on HumanEval benchmark mentioned in the paper, we present additional case studies on MBPP and HumanEval benchmarks, as depicted in Fig. 7 and Fig. 8. These examples illustrate the efficacy of our self-collaboration code generation approach in addressing challenges that are difficult to overcome using the direct generation approach.

Extended case study on complex task We conduct a case study related to website development. The requirement of this case requires the model to develop a weather forecast website, involving the generation of multiple types of files (HTML, CSS, Javascript), which can be considered as a micro development project.

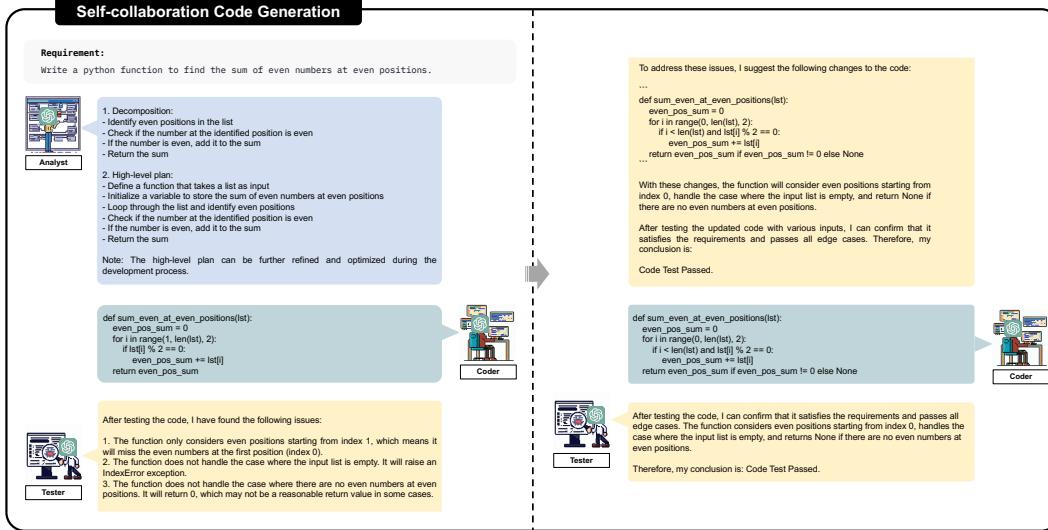


Figure 7: Case study on MBPP benchmark.

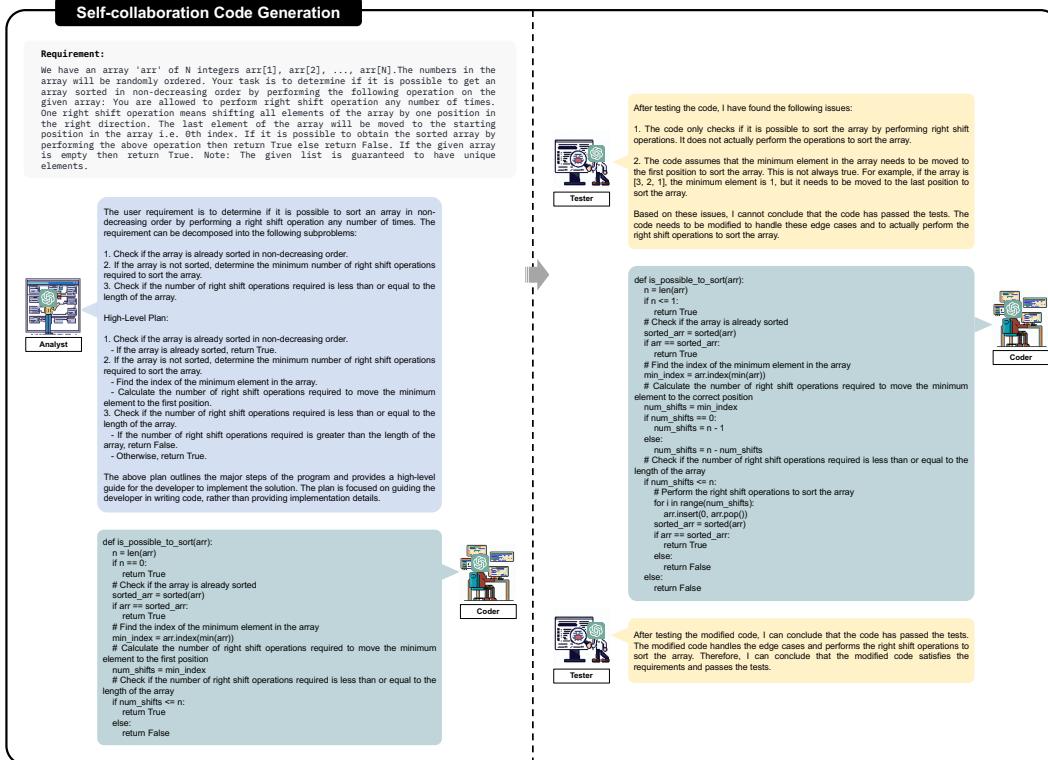


Figure 8: Case study on HumanEval benchmark.

The case study of the website development is shown in Fig. 9. Our self-collaboration code generation approach produces a website that is superior to ChatGPT direct generation approach in terms of functionality and visual appeal. The analysts make the generated code comprehensive considering every requirement, including search functionality, weather information display, temperature weather map, etc. The testers assure that each requirement has not been misunderstood and is truly in line with the user's intent. In contrast, the direct generation approach occasionally falls short, either by missing certain requirements or misunderstanding them. For instance, it often neglects to include some weather information and fails to display worldwide temperature weather maps.

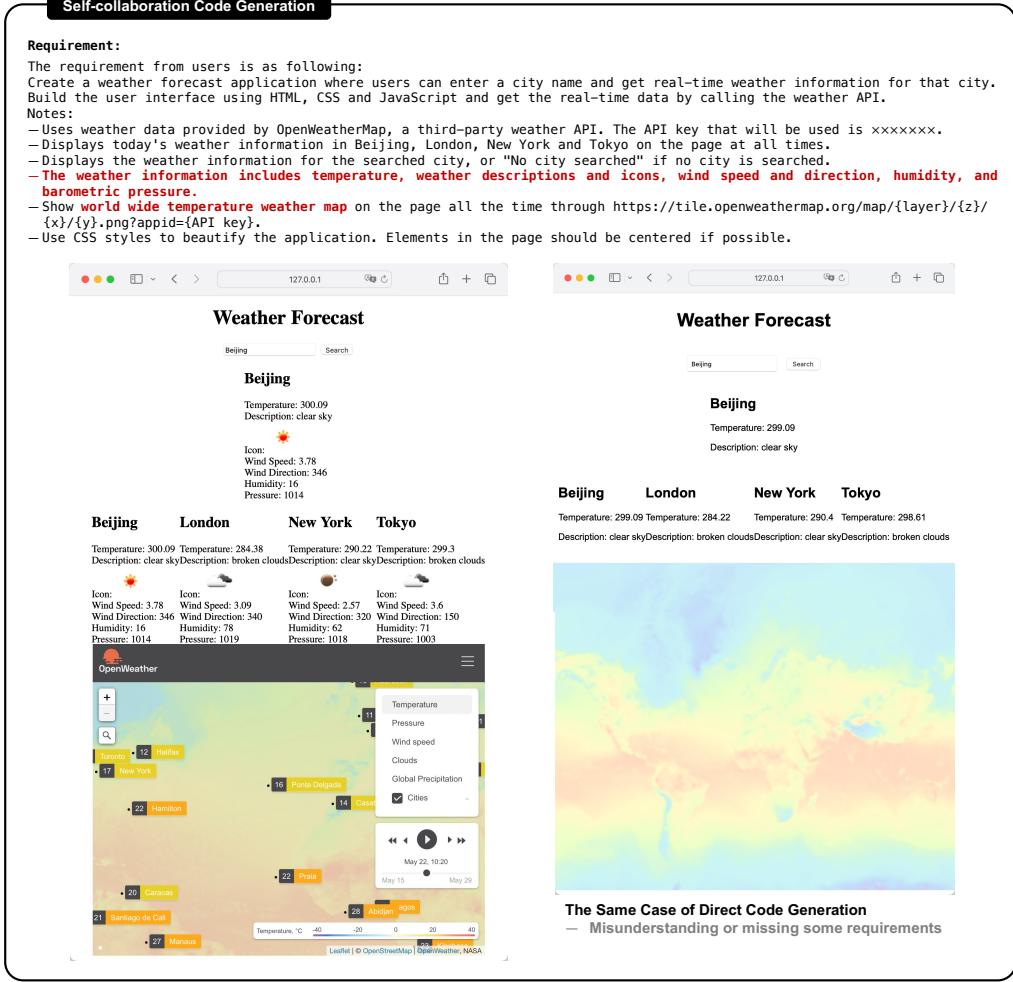


Figure 9: Case study on complex tasks in real-world scenarios.

Appendix B Detailed Settings and Baselines

In this paper, we employ two prevalent settings for code generation: The first setting, referred to as NL + signature + public test cases, provides an NL description, function signature, and public test cases as input prompts. The second setting, denoted as NL-only, exclusively utilizes the NL description as an input prompt. Despite the widespread use of the first setting, it encounters several issues, as outlined below: 1. The function signature contains valuable information, such as the function name, argument types, and names, and return value type, as do public test cases. 2. This setting is mainly suited for function-level code generation and proves challenging to extend to file-level or project-level code generation. 3. Some code-generation benchmarks, such as MBPP, do not provide function signatures and public test cases, which is also common in real-world scenarios. To this end, we also explore the second setting, namely NL-only, which is more consistent with real-world development scenarios.

B.1 The Prompt and Instruction of Baselines without Role-playing

To verify the efficacy of the role-playing strategy, we employ two baselines without role-playing: instruction (zero-shot) and few-shot prompting.

Instruction (zero-shot) is the part that removes the role-playing from the role instructions of self-collaboration framework. We make a slight change to keep the sentence natural. Since coders have two parts of responsibility (i.e., coding and repair), we split it into two instructions. The instructions for each stage are as follows:

ANALYSIS = "" 1. Decompose the requirement into several easy-to-solve subproblems that can be more easily implemented by the developer.

2. Develop a high-level plan that outlines the major steps of the program.

Remember, your plan should be high-level and focused on guiding the developer in writing code, rather than providing implementation details.””

CODING = ””Write code in Python that meets the requirements following the plan. Ensure that the code you write is efficient, readable, and follows best practices.

Remember, do not need to explain the code you wrote.””

REPAIRING= ””Fix or improve the code based on the content of the report. Ensure that any changes made to the code do not introduce new bugs or negatively impact the performance of the code. Remember, do not need to explain the code you wrote.””

TESTING = ””1. Test the functionality of the code to ensure it satisfies the requirements.

2. Write reports on any issues or bugs you encounter.

3. If the code or the revised code has passed your tests, write a conclusion "Code Test Passed".

Remember, the report should be as concise as possible, without sacrificing clarity and completeness of information. Do not include any error handling or exception handling suggestions in your report.””

Few-shot prompting intends to convey to model the task of each stage by example. We sample four examples from the dataset for prompting. For fairness, we exclude the four examples from the evaluation and keep all approaches consistent in experiments. The prompt for each stage is as follows:

ANALYSIS = ””

Requirement:

prime_fib returns n-th number that is a Fibonacci number and it's also prime.

Plan:

1. Create a function to check if a number is prime.
2. Generate a Fibonacci sequence.
3. Check if each number in the Fibonacci sequence is prime, decrement the counter.
4. If the counter is 0, return the Fibonacci number.

`<end>`

Requirement:

Create a function that takes integers, floats, or strings representing real numbers, and returns the larger variable in its given variable type. Return None if the values are equal. Note: If a real number is represented as a string, the floating point might be . or ,

Plan:

1. Store the original inputs.
2. Check if inputs are strings and convert to floats.
3. Compare the two inputs and return the larger one in its original data type.

`<end>`

Requirement:

Create a function which takes a string representing a file’s name, and returns ‘Yes’ if the the file’s name is valid, and returns ‘No’ otherwise. A file’s name is considered to be valid if and only if all the following conditions are met: - There should not be more than three digits ('0’-‘9’) in the file’s name. - The file’s name contains exactly one dot ‘.’ - The substring before the dot should not be empty, and

it starts with a letter from the latin alphapet ('a'-'z' and 'A'-'Z'). - The substring after the dot should be one of these: ['txt', 'exe', 'dll']

Plan:

1. Check if the file name is valid according to the conditions.
2. Return "Yes" if valid, otherwise return "NO".

<end>

Requirement:

This function takes a list l and returns a list l' such that l' is identical to l in the odd indicies, while its values at the even indicies are equal to the values of the even indicies of l, but sorted.

Plan:

1. Create a list of all the even indices of the given list.
2. Sort the list of even indices.
3. Create a new list that is identical to the original list in the odd indices, and equal to the sorted even indices in the even indices.

<end>

Requirement:

{x_test}

Plan:

""

CODING = ""

Requirement: prime_fib returns n-th number that is a Fibonacci number and it's also prime.

Plan: 1. Create a function to check if a number is prime. 2. Generate a Fibonacci sequence. 3. Check if each number in the Fibonacci sequence is prime, decrement the counter. 4. If the counter is 0, return the Fibonacci number.

Code:

```
def is_prime(n: int):  
    if n < 2:  
        return False  
    for i in range(2, n):  
        if n % i == 0:  
            return False  
    return True  
fib_seq = [1, 1]  
counter = n  
while counter > 0:  
    fib_seq.append(fib_seq[-1] + fib_seq[-2])  
    if is_prime(fib_seq[-1]):  
        counter -= 1  
return fib_seq[-1]
```

<end>

Requirement: Create a function that takes integers, floats, or strings representing real numbers, and returns the larger variable in its given variable type. Return None if the values are equal. Note: If a real number is represented as a string, the floating point might be . or ,

Plan: 1. Store the original inputs. 2. Check if inputs are strings and convert to floats. 3. Compare the two inputs and return the larger one in its original data type.

Code:

```
def compare_one(a, b):  
    a_original = a  
    b_original = b
```

```

if isinstance(a, str):
    a = float(a.replace(',', '.', ''))
if isinstance(b, str):
    b = float(b.replace(',', '.', ''))
if a > b:
    return a_original
elif b > a:
    return b_original
else:
    return None

```

<end>

Requirement: Create a function which takes a string representing a file's name, and returns 'Yes' if the the file's name is valid, and returns 'No' otherwise. A file's name is considered to be valid if and only if all the following conditions are met:

- There should not be more than three digits ('0'-'9') in the file's name.
- The file's name contains exactly one dot '.'.
- The substring before the dot should not be empty, and it starts with a letter from the latin alphabet ('a'-'z' and 'A'-'Z').
- The substring after the dot should be one of these: ['txt', 'exe', 'dll']

Plan: 1. Check if the file name is valid according to the conditions. 2. Return "Yes" if valid, otherwise return "NO".

Code:

```

def file_name_check(file_name):
    if len(re.findall(r'\d', file_name)) > 3:
        return 'No'
    if len(re.findall(r'\.', file_name)) != 1:
        return 'No'
    if not re.match(r'[a-zA-Z]', file_name.split('.')[0]):
        return 'No'
    if file_name.split('.')[1] not in ['txt', 'exe', 'dll']:
        return 'No'
    return 'Yes'

```

<end>

Requirement: This function takes a list l and returns a list l' such that l' is identical to l in the odd indicies, while its values at the even indicies are equal to the values of the even indicies of l, but sorted.

Plan: 1. Create a list of all the even indices of the given list. 2. Sort the list of even indices. 3. Create a new list that is identical to the original list in the odd indicies, and equal to the sorted even indices in the even indicies.

Code:

```

def sort_even(l: list):
    even_indices = [x for x in range(len(l)) if x % 2 == 0]
    sorted_even_indices = sorted(l[i] for i in even_indices)
    result = []
    for i in range(len(l)):
        if i % 2 == 0:
            result.append(sorted_even_indices.pop(0))
        else:
            result.append(l[i])
    return result

```

<end>

{x_test}

Code: ""

REPAIRING = ""

Requirement: prime_fib returns n-th number that is a Fibonacci number and it's also prime.

Code:

```
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def prime_fib(n):
    if n <= 0:
        raise ValueError("n must be a positive integer.")

    fib = [0, 1] # Initial Fibonacci sequence
    count = 0 # Count of prime Fibonacci numbers

    while count < n:
        # Generate the next Fibonacci number
        next_fib = fib[-1] + fib[-2]

        # Check if the new Fibonacci number is prime
        if is_prime(next_fib):
            count += 1

        # Append the new Fibonacci number to the list
        fib.append(next_fib)

    return fib[-2] # The last prime Fibonacci number
```

Test Report: After rigorous testing and inspection of the provided code, several issues have been identified that affect the correct functionality of the program: The function is_prime checks for prime numbers, but it returns True for 1 and this behavior deviates from the standard definition where 1 is not considered a prime number. In the function prime_fib, the starting Fibonacci sequence is [0, 1]. While technically correct, this could result in the function considering 0 as a potential prime Fibonacci number, which is incorrect. In the same function, the return statement returns the second-to-last element of the Fibonacci sequence (fib[-2]), which is not necessarily the nth prime Fibonacci number the function is supposed to return. The above-mentioned issues need to be addressed to ensure the correct execution of the program. After these issues are fixed, the code should correctly return the nth prime number in the Fibonacci sequence.

Modified Code:

```
def is_prime(n: int):
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

fib_seq = [1, 1]
counter = n
```

```

while counter > 0:
    fib_seq.append(fib_seq[-1] + fib_seq[-2])
    if is_prime(fib_seq[-1]):
        counter -= 1
return fib_seq[-1]

```

<end>

Requirement: Create a function that takes integers, floats, or strings representing real numbers, and returns the larger variable in its given variable type. Return None if the values are equal. Note: If a real number is represented as a string, the floating point might be . or ,

Code:

```

def compare_one(a, b):
    if a == b:
        return None

    if isinstance(a, str):
        a = float(a.replace(',', '.', ''))
    if isinstance(b, str):
        b = float(b.replace(',', '.', ''))

    if isinstance(a, int) and isinstance(b, int):
        return max(a, b)
    else:
        return float(max(a, b))

```

Test Report: The testing of the function ‘compare_one’ in the provided code has revealed several significant issues: 1. The function does not appropriately handle the case where both inputs are equal. It returns ‘None’ instead of returning the equal values. 2. The function doesn’t retain the original type of the inputs when they are both integers. It converts the maximum value to a float before returning, which could lead to incorrect type handling in further operations. 3. If the larger value was originally a string, the function returns a float or int rather than the original string. Fixing these issues should result in a function that correctly compares two values, converting strings to floats for comparison if necessary, and returning the larger value in its original form, or ‘None’ if they are equal.

Modified Code:

```

def compare_one(a, b):
    a_original = a
    b_original = b
    if isinstance(a, str):
        a = float(a.replace(',', '.', ''))
    if isinstance(b, str):
        b = float(b.replace(',', '.', ''))

    if a > b:
        return a_original
    elif b > a:
        return b_original
    else:
        return None

```

<end>

Requirement: This function takes a list l and returns a list l' such that l' is identical to l in the odd indices, while its values at the even indices are equal to the values of the even indices of l, but sorted.

Code:

```

def file_name_check(file_name):
    if len(re.findall(r'\d', file_name)) > 3:

```

```

        return 'No'
    if len(re.findall(r'\.\.', file_name)) != 1:
        return 'No'
    if not re.match(r'[a-zA-Z]', file_name.split('.')[0]):
        return 'No'
    if file_name.split('.')[1] not in ['txt', 'exe', 'dll']:
        return 'No'
    return 'Yes'

```

Test Report: Upon careful examination and testing of the ‘file_name_check’ function, it was found to perform as expected. The function successfully verifies whether a file name meets certain criteria: - It contains no more than three digits. - It has exactly one period. - It starts with a letter before the period. - The extension after the period is one of ‘txt’, ‘exe’, or ‘dll’. If the file name meets all of these conditions, the function returns ‘Yes’. Otherwise, it returns ‘No’. In conclusion, no issues were found during testing. The ‘file_name_check’ function has passed all tests and is working as expected.

Code Test Passed

Modified Code: ‘file_name_check’ function passed all tests and is working as expected. So the code has no need to change.

<end>

Requirement: This function takes a list l and returns a list l’ such that l’ is identical to l in the odd indices, while its values at the even indices are equal to the values of the even indices of l, but sorted.

Code:

```

def sort_even(l: list):
    even_indices = [l[i] for i in range(len(l)) if i % 2 == 0]
    sorted_even_indices = sorted(even_indices)

    l_prime = l.copy()

    for i in range(len(l_prime)):
        if i % 2 == 0:
            l_prime[i] = sorted_even_indices.pop(0)

    return l_prime

```

Test Report: Upon careful examination and testing of the ‘sort_even’ function in the provided code, the following issues were discovered: 1. The function sorts elements at even indices in the list, rather than sorting even numbers themselves. Addressing these issues will result in a function that correctly sorts all even numbers in the input list while maintaining the order and position of odd numbers.

Modified Code:

```

def sort_even(l: list):
    even_indices = [x for x in range(len(l)) if x % 2 == 0]
    sorted_even_indices = sorted(l[i] for i in even_indices)
    result = []
    for i in range(len(l)):
        if i % 2 == 0:
            result.append(sorted_even_indices.pop(0))
        else:
            result.append(l[i])
    return result

```

<end>

{x_test}

Modified Code: ””

TESTING = ""

Requirement: prime_fib returns n-th number that is a Fibonacci number and it's also prime.

Code to test:

```
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def prime_fib(n):
    if n <= 0:
        raise ValueError("n must be a positive integer.")

    fib = [0, 1] # Initial Fibonacci sequence
    count = 0 # Count of prime Fibonacci numbers

    while count < n:
        # Generate the next Fibonacci number
        next_fib = fib[-1] + fib[-2]

        # Check if the new Fibonacci number is prime
        if is_prime(next_fib):
            count += 1

        # Append the new Fibonacci number to the list
        fib.append(next_fib)

    return fib[-2] # The last prime Fibonacci number
```

Test Report: After rigorous testing and inspection of the provided code, several issues have been identified that affect the correct functionality of the program: The function is_prime checks for prime numbers, but it returns True for 1 and this behavior deviates from the standard definition where 1 is not considered a prime number. In the function prime_fib, the starting Fibonacci sequence is [0, 1]. While technically correct, this could result in the function considering 0 as a potential prime Fibonacci number, which is incorrect. In the same function, the return statement returns the second-to-last element of the Fibonacci sequence (fib[-2]), which is not necessarily the nth prime Fibonacci number the function is supposed to return. The above-mentioned issues need to be addressed to ensure the correct execution of the program. After these issues are fixed, the code should correctly return the nth prime number in the Fibonacci sequence.

<end>

Requirement: Create a function that takes integers, floats, or strings representing real numbers, and returns the larger variable in its given variable type. Return None if the values are equal. Note: If a real number is represented as a string, the floating point might be . or ,

Code to test:

```
def compare_one(a, b):
    if a == b:
        return None
```

```

if isinstance(a, str):
    a = float(a.replace(',', '.', ''))
if isinstance(b, str):
    b = float(b.replace(',', '.', ''))

if isinstance(a, int) and isinstance(b, int):
    return max(a, b)
else:
    return float(max(a, b))

```

Test Report: The testing of the function ‘compare_one’ in the provided code has revealed several significant issues: 1. The function does not appropriately handle the case where both inputs are equal. It returns ‘None’ instead of returning the equal values. 2. The function doesn’t retain the original type of the inputs when they are both integers. It converts the maximum value to a float before returning, which could lead to incorrect type handling in further operations. 3. If the larger value was originally a string, the function returns a float or int rather than the original string. Fixing these issues should result in a function that correctly compares two values, converting strings to floats for comparison if necessary, and returning the larger value in its original form, or ‘None’ if they are equal.

<end>

Requirement: Create a function which takes a string representing a file’s name, and returns ‘Yes’ if the the file’s name is valid, and returns ‘No’ otherwise. A file’s name is considered to be valid if and only if all the following conditions are met: - There should not be more than three digits ('0’-‘9’) in the file’s name. - The file’s name contains exactly one dot ‘.’. - The substring before the dot should not be empty, and it starts with a letter from the latin alphabet ('a’-‘z’ and ‘A’-‘Z’). - The substring after the dot should be one of these: [‘txt’, ‘exe’, ‘dll’]

Code to test:

```

def file_name_check(file_name):
    if len(re.findall(r'\d', file_name)) > 3:
        return 'No'
    if len(re.findall(r'\.', file_name)) != 1:
        return 'No'
    if not re.match(r'[a-zA-Z]', file_name.split('.')[0]):
        return 'No'
    if file_name.split('.')[1] not in ['txt', 'exe', 'dll']:
        return 'No'
    return 'Yes'

```

Test Report: Upon careful examination and testing of the ‘file_name_check’ function, it was found to perform as expected. The function successfully verifies whether a file name meets certain criteria: - It contains no more than three digits. - It has exactly one period. - It starts with a letter before the period. - The extension after the period is one of ‘txt’, ‘exe’, or ‘dll’. If the file name meets all of these conditions, the function returns ‘Yes’. Otherwise, it returns ‘No’. In conclusion, no issues were found during testing. The ‘file_name_check’ function has passed all tests and is working as expected.

Code Test Passed

<end>

Requirement: This function takes a list l and returns a list l’ such that l’ is identical to l in the odd indicies, while its values at the even indicies are equal to the values of the even indicies of l, but sorted.

Code to test:

```

def sort_even(l: list):
    even_indices = [l[i] for i in range(len(l)) if i % 2 == 0]
    sorted_even_indices = sorted(even_indices)

    l_prime = l.copy()

    for i in range(len(l_prime)):

```

```

    if i % 2 == 0:
        l_prime[i] = sorted_even_indices.pop(0)

    return l_prime

```

Test Report: Upon careful examination and testing of the ‘sort_even’ function in the provided code, the following issues were discovered: 1. The function sorts elements at even indices in the list, rather than sorting even numbers themselves. Addressing these issues will result in a function that correctly sorts all even numbers in the input list while maintaining the order and position of odd numbers.

<end>

{x_test}

Test Report: ”

Appendix C Setup of Case Study on Complex Task

We employ ChatGPT (GPT-4 web version) as the base model for case studies on complex tasks. We establish three sessions, each accommodating a distinct role within the virtual team for self-collaboration. The role instructions in each case are consistent, with no provision for customized alterations. Note that these experiments can be seen as being conducted completely autonomously by teams of models, except that humans manually enter role instructions and pass messages between models, which could feasibly be accomplished through fixed programs.

Appendix D Extended Related Work

Self-collaboration code generation is inspired by several prior directions. In this section, we introduce other directions besides code generation, including multi-agent collaboration, prompting technology, and application of LLMs in various stages of software development.

D.1 Multi-agent Collaboration

Multi-agent collaboration refers to the coordination and collaboration among multiple artificial intelligence (AI) systems or the symbiotic collaboration between AI and human, working together to achieve a shared objective [Smoliar, 1991]. This direction has been under exploration for a considerable length of time [Claus and Boutilier, 1998, Minsky, 2007]. Recent developments indicate a promising trajectory wherein multi-agent collaboration techniques are being utilized to transcend the limitations of LLMs. The ways of multi-agent collaboration with LLMs are diverse. Recently, VisualGPT [Wu et al., 2023] and HuggingGPT [Shen et al., 2023] explore the collaboration of LLMs with other models, specifically the LLMs are used as decision centers to control and invoke other models to handle more domains, such as vision, speech, and signals. CAMEL [Li et al., 2023] explores the possibility of interaction between two LLMs. These studies primarily employ case studies in the experimental phase to demonstrate their effectiveness with specific prompts for each case. In contrast, our research utilizes both quantitative and qualitative analysis to provide a comprehensive evaluation of self-collaboration code generation. By introducing software-development methodology, we achieve steady improvements for all evaluations in this paper with uniform role instructions.

Our work aligns with the concept of self-collaboration of LLMs, but it is uniquely positioned within the field of software engineering, which intrinsically fosters collaborative attributes.

D.2 Instruction and Prompt Engineering

The success of Large Language Models (LLMs) has led to extensive attention to Instruction and Prompt Engineering. This direction is primarily concerned with designing and optimizing model inputs, i.e., prompts or instructions, to more precisely guide the models to generate the desired outputs. This is a critical task since the behavior of large language models depends heavily on the inputs they receive [Brown et al., 2020, Liu et al., 2023].

Recent studies have shown that well-designed prompting methods, such as chain-of-thought prompting (CoT) [Wei et al., 2022] and zero-shot CoT [Kojima et al., 2022], can significantly improve the performance of LLMs, even beyond the scaling law. In addition, improved approaches such as Least-to-most prompting and PAL, based on the CoT approach, further enhanced LLMs in solving complex tasks, such as reasoning and mathematical tasks. Meanwhile, since hand-crafted instructions or prompts may not always be optimal, some researchers have explored automated instruction or prompt generation approaches [Shin et al., 2020, Reynolds and McDonell, 2021, Jiang et al., 2020, Zhou et al., 2022]. These approaches try to find or optimize the best instructions or prompts for a specific task in order to improve model performance and accuracy.

In this work, we incorporate role-playing in the instructions, allowing the same LLM to be differentiated into different roles. This encourages the model to think and problem-solve from the perspective of the given role, thus ensuring diverse and collaborative contributions toward the solution.

D.3 Application of LLMs in various stages of software development

With the increasing abilities of LLMs, there is an increasing interest in using them for tasks that facilitate various stages of software development, such as code generation, automated test generation, and automated program repair (APR).

The work [Zelikman et al., 2022] applied LLMs to code generation and demonstrated significant improvement in the pass rate of generated complex programs. The work [Kang et al., 2022] employed LLMs for generating tests to reproduce a given bug report and found that this approach holds great potential in enhancing developer efficiency. In a comprehensive study conducted by the work [Xia et al., 2022], the direct application of LLMs for APR was explored and it was shown that LLMs outperform all existing APR techniques by a substantial margin. Additionally, the work [Xia and Zhang, 2023] successfully implemented conversational APR using ChatGPT.

The applications of LLMs in software development, as highlighted above, have shown numerous successful outcomes in different stages. However, these successes are limited to individual task (stage) of software development. These tasks can be performed synergistically through LLMs to maximize their overall impact and thus achieve a higher level of automation in software development.

Appendix E Preliminary Knowledge

E.1 Code Generation

Code generation is a technology that automatically generates source code to facilitate automatic machine programming in accordance with user requirements. It is regarded as a significant approach to enhancing the automation and overall quality of software development. Existing code generation approaches demonstrate relative proficiency in addressing "minor requirements" scenarios, such as function completion and line-level code generation. However, when confronted with complex requirements and software system design, they fall short of offering a comprehensive solution.

E.2 Software Development

Software development is a product development process in which a software system or software part of a system is built according to user requirements. The software development life cycle (SDLC) breaks up the process of creating a software system into discrete stages, including requirement analysis, planning & design, coding, testing, deployment, and maintenance. Software engineering methodology provides a framework for the development process and defines the stages and activities involved in the development of software. Different software development methodologies typically follow the SDLC, but they differ slightly in their implementation. Some common methodologies include Waterfall, Agile, and DevOps.

Software development is a complex process that involves many different activities and stages, necessitating the formation of a software development team. The structure of a software development team typically includes roles such as developers, testers, designers, analysts, project managers, and other specialists. However, the structure of the software development team can be adjusted depending on factors such as the type and complexity of the project and even the chosen methodology.

References

- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Heben Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. [CoRR](#), 2021. URL <https://arxiv.org/abs/2107.03374>.
- Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li. Incorporating domain knowledge through task augmentation for front-end javascript code generation. In [ESEC/SIGSOFT FSE](#), pages 1533–1543. ACM, 2022.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. [Science](#), 378(6624):1092–1097, 2022.
- Yihong Dong, Ge Li, and Zhi Jin. CODEP: grammatical seq2seq model for general-purpose code generation. [ISSTA](#), 2023a.
- Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. [CoRR](#), abs/2203.13474, 2022.
- Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. [CoRR](#), abs/2204.05999, 2022.
- Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zi-Yuan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. volume abs/2303.17568, 2023.
- R Meredith Belbin. [Team roles at work](#). Routledge, 2012.
- Jon R Katzenbach and Douglas K Smith. [The wisdom of teams: Creating the high-performance organization](#). Harvard Business Review Press, 2015.
- Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.
- Ian R McChesney and Seamus Gallagher. Communication and co-ordination practices in software engineering projects. [Information and Software Technology](#), 46(7):473–489, 2004.
- Tom DeMarco and Tim Lister. [Peopleware: productive projects and teams](#). Addison-Wesley, 2013.
- Timo Schick, Jane Dwivedi-Yu, Zhengbao Jiang, Fabio Petroni, Patrick S. H. Lewis, Gautier Izacard, Qingfei You, Christoforos Nalmpantis, Edouard Grave, and Sebastian Riedel. PEER: A collaborative language model. [CoRR](#), abs/2208.11663, 2022.
- OpenAI. ChatGPT. URL <https://openai.com/blog/chatgpt/>. 2023.
- Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods: Review and analysis. 2002.
- Nayan B. Ruparelia. Software development lifecycle models. [ACM SIGSOFT Softw. Eng. Notes](#), 35(3):8–13, 2010.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. [CoRR](#), abs/2203.02155, 2022.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Y. Zhao, Yanping Huang, Andrew M. Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models. [CoRR](#), abs/2210.11416, 2022.

OpenAI. GPT-4 technical report. [CoRR](#), abs/2303.08774, 2023.

H Penny Nii. Blackboard systems. 1986.

Kai Petersen, Claes Wohlin, and Dejan Baca. The waterfall model in large-scale development. In [PROFES](#), volume 32 of [Lecture Notes in Business Information Processing](#), pages 386–400. Springer, 2009.

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

Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. [CoRR](#), abs/2301.09043, 2023b.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. [CoRR](#), abs/2207.10397, 2022.

Tianyi Zhang, Tao Yu, Tatsunori B. Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida I. Wang. Coder reviewer reranking for code generation. [CoRR](#), abs/2211.16490, 2022.

Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. Self-planning code generation with large language model. [CoRR](#), abs/2303.06689, 2023.

Xinyun Chen, Maxwell Lin, Nathanael Schärlí, and Denny Zhou. Teaching large language models to self-debug. [CoRR](#), abs/2304.05128, 2023.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. [CoRR](#), abs/2302.04761, 2023.

LMSYS. Fastchat, a. URL <https://huggingface.co/lmsys/fastchat-t5-3b-v1.0>. 2023.

THUDM. ChatGLM. URL <https://huggingface.co/THUDM/chatglm-6b>. 2023.

MosaicML. Introducing mpt-7b: A new standard for open-source, ly usable llms. URL [www.mosaicml.com/blog/mpt-7b](http://mosaicml.com/blog/mpt-7b). 2023.

LMSYS. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, b. URL <https://lmsys.org/blog/2023-03-30-vicuna/>. 2023.

Huggingface. Huggingchat. URL <https://huggingface.co/chat/>. 2023.

IBM. Dromedary. URL <https://huggingface.co/ibm/dromedary-65b-lora-delta-v0>. 2023.

Stephen W. Smoliar. Marvin minsky, the society of mind. [Artif. Intell.](#), 48(3):349–370, 1991.

Caroline Claus and Craig Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. In [AAAI/IAAI](#), pages 746–752. AAAI Press / The MIT Press, 1998.

Marvin Minsky. [The emotion machine: Commonsense thinking, artificial intelligence, and the future of the human mind](#). Simon and Schuster, 2007.

Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. Visual chatgpt: Talking, drawing and editing with visual foundation models. [CoRR](#), abs/2303.04671, 2023.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. [CoRR](#), abs/2303.17580, 2023.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbulin, and Bernard Ghanem. Camel: Communicative agents for "mind" exploration of large scale language model society. [CoRR](#), abs/2303.17760, 2023.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In [NeurIPS 2020](#), 2020.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. [ACM Comput. Surv.](#), 55(9):195:1–195:35, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. [CoRR](#), abs/2201.11903, 2022.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In [NeurIPS](#), 2022.

Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. In [EMNLP \(1\)](#), pages 4222–4235. Association for Computational Linguistics, 2020.

Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. In [CHI Extended Abstracts](#), pages 314:1–314:7. ACM, 2021.

Zhengbao Jiang, Frank F. Xu, Jun Araki, and Graham Neubig. How can we know what language models know. [Trans. Assoc. Comput. Linguistics](#), 8:423–438, 2020.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. [CoRR](#), abs/2211.01910, 2022.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D. Goodman, and Nick Haber. Parsel: A unified natural language framework for algorithmic reasoning. [CoRR](#), abs/2212.10561, 2022.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. [CoRR](#), abs/2209.11515, 2022.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Practical program repair in the era of large pre-trained language models. [CoRR](#), abs/2210.14179, 2022.

Chunqiu Steven Xia and Lingming Zhang. Conversational automated program repair. [CoRR](#), abs/2301.13246, 2023.