Self-collaboration Code Generation via ChatGPT

YIHONG DONG*, XUE JIANG*, ZHI JIN, and GE LI†, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China

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 LLM agents 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 LLM 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 the base LLM agent. Moreover, we showcase that self-collaboration could potentially enable LLMs to efficiently handle complex repository-level tasks that are not readily solved by the single LLM agent.

CCS Concepts: • **Software and its engineering → Software creation and management**; • **Computing methodologies → Artificial intelligence**.

Additional Key Words and Phrases: Code Generation, Large Language Models, Multi-Agent Collaboration, Software Development.

ACM Reference Format:
Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (May 2023), 38 pages. https://doi.org/XXXXXXX.XXXXXXX

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

*Equal Contribution
†Corresponding author
[0]https://github.com/YihongDong/Self-collaboration-Code-Generation

Authors' address: Yihong Dong, dongyh@stu.pku.edu.cn; Xue Jiang, jiangxue@stu.pku.edu.cn; Zhi Jin, zhijin@pku.edu.cn; Ge Li, lige@pku.edu.cn, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China.

1049-331X/2023/5-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

both academic and industrial domains [8, 16, 30, 49]. In particular, LLMs have achieved excellent performance and demonstrate promising potential on code generation tasks [15, 18, 38, 64].

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, has been theorized to play an important role in dealing with complexity, as posited in both teamwork theory [5, 26] and software engineering practice [4, 13, 34]. 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 [48]. 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 [39], 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.

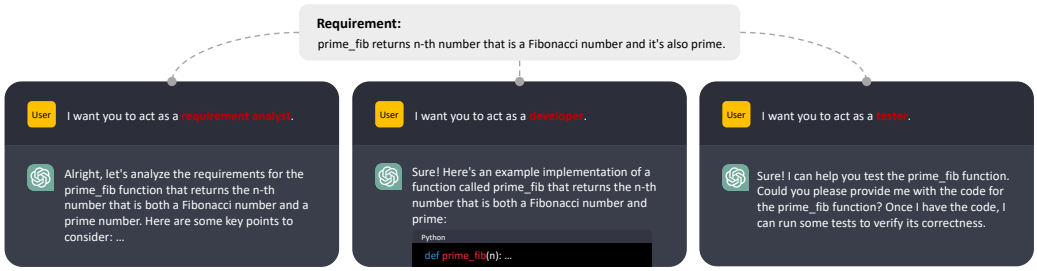

Fig. 1. An example of role-playing. Through role-playing, LLM transforms into an expert within a specific domain, delivering a professional-perspective response to the same requirement.

To 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.

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

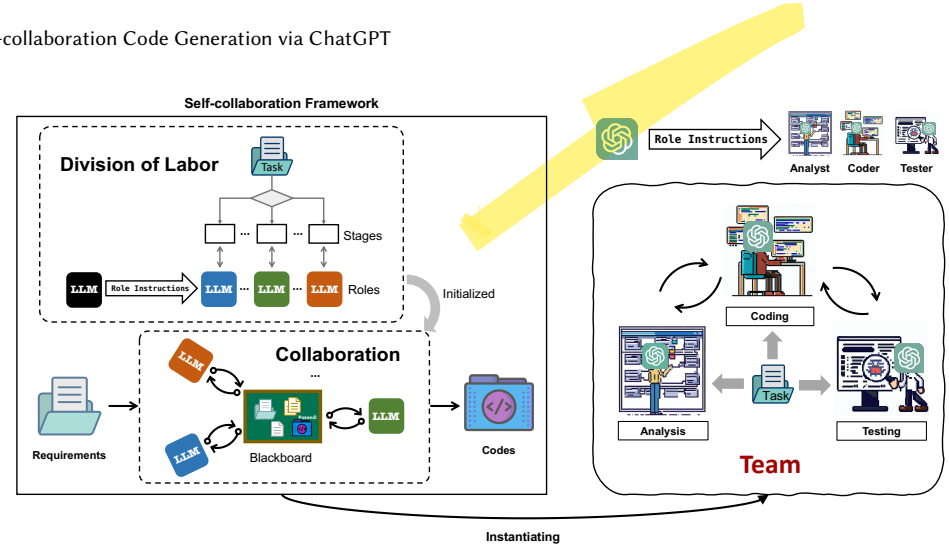

Fig. 2. Self-collaboration framework for code generation and its instance.

complexity and improving software quality [1, 46]. Following SDM, we instantiate an elementary team composed of three roles (i.e., analyst, coder, and tester) to achieve this goal. These roles adhere to an SDM-defined 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 creates or improves code based on the plans or the tester's feedback; the tester compiles test reports based on the coder's outcome and documents any issues found during testing. We employ three ChatGPT[1] 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 LLM agents to collaborate with each other to generate code for complex requirements.
(2) Following software-development methodology, we instantiate an elementary team, which comprises three LLM 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 significant improvements compared to the single LLM agent on multiple code-generation benchmarks.
(4) In some real-world scenarios, self-collaboration code generation exhibits notable effectiveness on more complex code generation tasks (such as repository-level code generation) that are challenging for the single LLM agent.

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 $\mathcal{T} : x \rightarrow y$.

[1]The ChatGPT referenced throughout our paper defaults to the GPT-3.5 version.

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.[2]

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 [11, 40, 41]. 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, a clear and unverbose task description in the instruction could lead to LLM's behavior being more 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 the role instruction only needs to be provided once at the initialization of each LLM agent to provide specific guidance on its behavior throughout subsequent interaction, thus enhancing the overall efficiency and clarity in 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 role, information, and format that each role interacts with in role instructions, which allows the whole process of collaboration to be well controlled. The collaboration part can be formalized as follows:

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

where s_t is the output of stage $\mathcal{S}_t$, $s_{\{<t\}}$ indicates the prerequisite-stage outputs of $\mathcal{S}_t$[3], and $R_{m(S_t)}$ represents the role corresponding to $\mathcal{S}_t$. 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 $\mathcal{S}_t$:

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

[2]In our self-collaboration framework, each stage is managed by a specific type of roles, but the number of roles within that type can vary. The order of the stages determines the sequence in which different types of roles are executed, but the roles within a single stage operate in parallel.

[3]Note that our self-collaboration framework can be parallelized if the relationship between stages $\{\mathcal{S}_i\}_{i=1}^{l}$ is not a straightforward linear relationship. In other words, if one stage does not depend on the results of another stage, then they can be executed in parallel. A specific scenario is that in a software development project including front-end, back-end, and database development, the analysis stage only needs to define the interface in advance and the corresponding coding stages can be carried out in parallel.

Algorithm 1 Pseudocode of self-collaboration framework.

Require: Requirement x, Task $\mathcal{T}$, and LLM $\mathcal{M}$.
Ensure: Output y.

```
# DOL Part
Initial Stages {S_i}_{i=1}^{l} according to task T.
Initial Roles {R_j}_{j=1}^{m} for LLMs M's based on stages {S_i}_{i=1}^{l} and role instructions.
# Collaboration Part
Initial blackboard B and index t.
repeat
    Obtain s_{<t} from B.
    Sample s_t via Eq. (1).
    Add s_t to B.
    Compute y_t via Eq. (2).
    Update y and t.
until End condition is satisfied
return y
```

where f is an update function. Once the end condition is satisfied[4], the final output y is derived. To coordinate collaboration between different roles, we set up a shared blackboard [37], from which each role exchanges the required information to accomplish their respective tasks s_t via Eq. (1). The pseudocode of our self-collaboration framework is outlined in Algorithm 1.

3 INSTANCE

We introduce the classic waterfall model [43] from software-development methodology 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 roles are assigned the following tasks:

Analyst. The goal of the analyst is to reduce the difficulty of coding by abstracting and decomposing the task from a high level, rather than delving into the details of the implementation. Given a requirement x, the analyst breaks x down into several easily solvable subtasks to facilitate the division of functional units and develops a high-level plan to guide the coder in writing the code.

Coder. As the central role of this team, the coder is responsible for writing the code, but its work is carried out with the assistance and supervision of the analyst and tester. 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 is responsible for inspecting the code and generating a test report on various aspects of functionality, readability, and maintainability to help the coder improve the quality of its code. Rather than directly introducing a compiler and test cases to execute the code, we use the model to simulate the testing process and produce test reports, thereby avoiding external efforts.

[4]The end condition is defined by prior knowledge, and an example can be found in the last paragraph of Section 3.

Role Instructions = **Team Description** + **User Requirment** + **Role Description**

Team Description	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.
User Requirment	The requirement from users is '{Requirment}'. *For example: {Requirment} = 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*
Role Description	**Coder:** 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: 1. 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. 2. 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. Remember, do not need to explain the code you wrote.

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

We customize role instructions for LLMs (exemplified by ChatGPT) to play the three roles. The role instruction includes not only the role description (role and its responsibilities) but also the team description and the user requirements, which will work together to initialize the ChatGPT agent, thereby setting the behavior of ChatGPT. An example of role instruction for coder is shown in Fig. 3. 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. When dealing with y_t composed of multiple components, it is recommended to ask LLMs to generate outputs in JSON format directly, which can reduce omissions and enhance the quality of generation outputs.

4 EVALUATION

We aim at answering the following research questions (RQs):

- **RQ1**: What is the performance of self-collaboration approach compared to the various baselines on public code-generation benchmarks?
- **RQ2**: What is the effect of roles in self-collaboration? Specifically, it can be divided into three questions: 1. What is the contribution of each role in the virtual team? 2. What is the effect of other virtual teams? 3. What is the effect of role-playing?
- **RQ3**: What is the performance of self-collaboration based on different LLMs, especially the most powerful LLM GPT-4?
- **RQ4**: What is the impact of interaction numbers for self-collaboration?
- **RQ5**: What are the results of more detailed analysis (specifically, error analysis and cost analysis) for self-collaboration?

- **RQ6**: How does self-collaboration work in repository-level software development scenarios and how does it perform?

4.1 Experiment Setup

4.1.1 Benchmarks. We perform a comprehensive evaluation on six code-generation benchmarks to demonstrate the efficacy of self-collaboration. In addition, we also collect several projects in real-world development environments to showcase the excellent performance of self-collaboration on more complex code generation tasks.

- **MBPP** (sanitized version) [3] 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** [8] consists of 164 handwritten programming tasks, proposed by OpenAI. Each task includes a function signature, NL description, use cases, function body, and several unit tests (average 7.7 per task).
- **MBPP-ET** and **HumanEval-ET** [14] 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.
- **APPS** [20] consists of problems collected from different open-access coding websites such as Codeforces, Kattis, and more. Problems are more complicated, as the average length of a problem is 293.2 words (about 5-25 times higher than preceding benchmarks). Each problem has multiple specifically designed test cases, and the average number of test cases is 21.2.
- **CoderEval** [61] contains four types of code in addition to the standalone code, i.e., Plib-depend, Class-depend, File-epend, and Project-depend, to simulate the dependencies in real-world development.

4.1.2 Baselines. In this paper, we compare self-collaboration with three kinds of baselines: LLMs customized for code, generalist LLMs, and prompting approaches.

I. LLMs customized for code *represent the LLMs specifically designed to solve code generation tasks and optimized on a large amount of code data, which helps to locate the performance level of our approach. This kind of baseline includes:*

- **AlphaCode (1.1B)** [30] is a transformer-based code generation model, which is trained on selected public codes before July 14, 2021, and can solve some basic code generation problems.
- **Incoder (6.7B)** [18] is a unified generation model that allows left-to-right code generation and code infilling/editing by the causal mask language modeling training objective.
- **CodeGeeX (13B)** [64] is a large-scale multilingual code generation model with 13 billion parameters, pre-trained on a large code corpus of more than 20 programming languages.
- **StarCoder (15.5B)** [29] is a Code LLM trained on permissively licensed data from GitHub, including from 80+ programming languages, Git commits, GitHub issues, and Jupyter notebooks.
- **CodeGen (16.1B)** [38] is a LLM trained on NL and programming data for conversation-based program synthesis. In this paper, we employ the CodeGen-Multi version.
- **PaLM Coder (540B)** [10] is finetuned from PaLM 540B on code, where PaLM uses an ML system, named Pathways, that enables highly efficient training of LLMs across thousands of accelerator chips.
- **CodeX (175B)** [8], also known as code-davinci-002, is fine-tuned from davinci 175B [6] on multilingual code data with code-completion tasks. CodeX is also the backbone model that powers Copilot [19] (a well-known commercial application).

- **CodeX (175B) + CodeT** [7] is a previous state-of-the-art (SOTA) approach before GPT-4. CodeT employs LLMs to automatically generate test cases for code samples. It executes code samples with these test cases and conducts a dual execution agreement, taking into account output consistency against test cases and output agreement among code samples.
- **CodeLlama (34B)** [45] is an open foundational model for code generation tasks, derived from continuous training and fine-tuning based on Llama 2 [55].

II. Generalist LLMs *represent the LLMs that are trained on data widely collected from Internet and show strong performance on a variety of tasks, which are used as the base model of our approach. This kind of baseline includes:*

- **ChatGPT** [39] is a sibling model to InstructGPT [42], which is trained to follow an instruction in a prompt and provide a detailed response. We access ChatGPT through OpenAI's API. Since ChatGPT receives regular updates, we employ a fixed version 'gpt-3.5-turbo-0301' as our base model, which will not receive updates, to minimize the risk of unexpected model changes affecting the results.
- **GPT-4** [40] is a large-scale, multimodal model which can accept image and text inputs and produce text outputs. GPT-4 exhibits human-level performance on various benchmarks.

III. Prompting approaches *here represent the general prompting approaches as well as those specifically designed for code generation, representing a category of our related work. This kind of baseline includes:*

- **Chain-of-thought (CoT)** [57] generates a chain of thought for each question and then generates the corresponding code. For CoT, we use the instruction "Let's think step by step." [27] to implement it.
- **Self-planning** [23] and **Self-debugging** [9] teach LLMs to perform planning and debugging with few-shot prompting. We use the prompts provided in their original papers to implement them.
- **Iter-improving** is a baseline approach proposed in this paper, which considers allowing the base model to continuously improve the generated code until it can no longer be modified. This approach is used to demonstrate that the effectiveness of our method is not solely due to multiple improvements to the output of LLMs. We use the instruction 'Please improve this code' to implement iter-improving, and the maximum number of iterations is set to 10.

4.1.3 Evaluation Metric. In this paper, we mainly focus on **Pass@1**: the probability that the model solves the problem in one attempt, since in real-world scenarios we usually only consider one generated code. We adopt the unbiased variant of Pass@1 [8] to measure the functional correctness of top-1 generated codes by executing test cases, which can be formulated as:

$$\text{Pass@1} = \mathop{\mathbb{E}}_{\text{Problems}} \left[\chi(\text{generated code}) \right]. \tag{3}$$

where $\mathop{\mathbb{E}}_{\text{Problems}}$ denotes the expectation of all problems and χ is an indicator function that outputs 1 if generated code passes all test cases of the corresponding problem, otherwise 0.

4.1.4 Implementation Details. For self-collaboration and all prompting approaches, we employ ChatGPT as the base model. We access ChatGPT using the 'gpt-3.5-turbo' API with fixed version '0301', which will not receive updates. For self-collaboration, the maximum number of interactions between roles is limited to 4. In all experiments, we set max tokens to 512 and temperature to 0 for code generation. We only reference the results reported in their original papers for AlphaCode (1.1B) [30] and PaLM Coder (540) [10], which are inaccessible to us. The results of all other baselines are

evaluated under the same settings of self-collaboration for fairness. All these results are comparable with the results reported in their original paper.

Table 1. Comparison of self-collaboration and baselines, where the green highlights indicate the improvements in comparison to ChatGPT (GPT-3.5).

Approach	HumanEval	HumanEval-ET	MBPP	MBPP-ET
LLMs Customized for Code				
AlphaCode (1.1B) [30]	17.1	-	-	-
Incoder (6.7B) [18]	15.2	11.6	17.6	14.3
CodeGeeX (13B) [64]	18.9	15.2	26.9	20.4
StarCoder (15.5B) [29]	34.1	25.6	43.6	33.4
CodeGen-Mono (16.1B) [38]	32.9	25.0	38.6	31.6
PaLM Coder (540B) [10]	36.0	-	47.0	-
CodeLlama (34B) [45]	48.2	36.8	55.9	42.1
CodeX (175B) [8]	47.0	31.7	58.1	38.8
CodeX (175B) + CodeT [7]	65.8	51.7	67.7	45.1
Generalist LLMs				
ChatGPT (GPT-3.5) [39]	57.3	42.7	52.2	36.8
GPT-4 [40]	67.6	50.6	68.3	52.2
Prompting Approaches				
Zero-shot CoT [57]	44.6	37.2	46.1	34.8
Self-planning [23]	65.2	48.8	58.6	41.5
Self-debugging [9]	61.6	45.8	60.1	52.3
Iter-improving	55.5	46.3	45.7	32.8
Ours	**74.4** (↑ 29.9%)	**56.1** (↑ 31.4%)	**68.2** (↑ 30.7%)	**49.5** (↑ 34.6%)

4.2 RQ1: Self-collaboration vs. Baselines

We compare self-collaboration with baselines in the 'NL + signature + use cases' setting[5], as shown in Table 1. Experimental results show that self-collaboration code generation based on ChatGPT (GPT-3.5), with a three-person team (including analysts, coders, and testers), achieves SOTA performance across the four code generation benchmarks. When compared to ChatGPT, 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 self-collaboration 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.

Compared to zero-shot CoT, Iter-improving, and two concurrent works (self-planning and self-debugging), self-collaboration also performs significantly better than these prompting baselines. It is worth mentioning that CoT is a successful prompting technique that solves the reasoning problem by generating intermediate reasoning steps, but CoT is not suitable for code generation [23, 53]. We find that the performance of Iter-improving is usually lower than the single LLM agent

[5]This setting provides requirements, function signature, and use cases as input to generate code. which is commonly used for baselines, so we also adopt this setting for comparison. However, using only requirements (NL-only) as input is more consistent with real-world development scenarios and more challenging. Therefore, in other experiments, we use the 'NL-only' setting if not specified. Detailed discussions can be found in Appendix B.

Table 2. The performance of self-collaboration code generation on APPS, where the green highlights indicate the improvements in comparison to ChatGPT (GPT-3.5).

Approach	Competition	Interview
CodeX (175B) + CodeT [7]	2.2	8.1
ChatGPT (GPT-3.5) [39]	1.2	6.8
Ours	**3.4** (↑ 183.3%)	**10.7** (↑ 57.4%)

with ChatGPT except for HumanEval-ET. This is because ChatGPT tends to optimize for edge test cases. While this helps in HumanEval-ET, it sometimes leads to unnecessary modifications of correct edge cases. Such adjustments might be safer in some scenarios, like throwing exceptions, but deviate from expected output.

We evaluate self-collaboration on a more algorithmically intricate benchmark, namely, APPS. This benchmark comprises three levels of difficulty: introductory, interview, and competition. In this experiment, we select the first 1000 code generation tasks from the most challenging two levels: interview and competition. Following the settings in CodeT [7], we use NL in APPS as a comment and then concatenate the signature '*def solution(stdin : str) → str :*' as input to the model. Experimental results shown in Table 2 indicate that self-collaboration enhances the performance of ChatGPT substantially and exceeds the performance of the previous SOTA approach CodeX (175B) + CodeT [7].

To evaluate self-collaboration on a more challenging and realistic benchmark, we conduct experiments on CoderEval [61] based on ChatGPT (GPT-3.5), following the settings in its paper. As shown in Table 3, self-collaboration code generation substantially outperforms the single ChatGPT, achieving relative improvements of 47.1% on Pass@1.

Table 3. The performance of self-collaboration code generation on CoderEval.

Approach	Standalone	Plib-depend	Class-depend	File-epend	Project-depend	Average
ChatGPT (GPT-3.5)	35.87	21.43	8.73	16.91	9.57	19.82
Ours	**47.63**	**38.1**	**21.82**	**20.59**	**13.04**	**29.14** (↑ 47.1%)

4.3 RQ2: The Effect of Roles in Self-collaboration

We investigated code generation relying solely on NL descriptions. In this setting, we explore the effect of roles in self-collaboration in three ways: 1) Conducting ablation studies with the three-roles team we assembled (analyst, coder, and tester). 2) Exploring alternative team configurations, i.e., a team incorporating a compiler, and a team of pair programming. The team with the compiler constructs the use cases in HumanEval as public test cases and obtains the execution results through compiler to help the tester generate reports. In the team with pair programming, we use a two-agent team of driver and observer for code generation. 3) Implementing the analysis, coding, and testing phases directly without role-playing to validate the role of role-playing. To implement these tasks, we used both instruction and few-shot prompting strategies. Instruction (zero-shot), which involves instructions with the role-playing component excised, and few-shot prompting, which provides examples demonstrating the tasks of analyzing, coding, and testing. Details of instructions and prompts are provided in Appendix F. The experimental results are shown in Table 4.

First, the 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

Table 4. Effectiveness of ChatGPT roles in self-collaboration code generation, where the green highlights indicate the improvements in comparison to Coder.

Roles	HumanEval	HumanEval-ET
Ablation of Roles		
Coder	45.1	35.3
+ Analyst	54.9 (↑ 21.8%)	45.2 (↑ 28.1%)
+ Tester	56.8 (↑ 26.0%)	45.2 (↑ 28.1%)
+ Analyst + Tester	**63.5** (↑ 40.8%)	**51.9** (↑ 47.1%)
Different Team Configurations		
Analyst + Coder + Tester + Compiler	64.6	50.0
Driver + Observer (Pair Programming)	57.4	44.6
The Role of Role-playing		
Few-shot Prompting (4-shot)	58.1	47.5
Instruction (zero-shot)	60.0	47.5
Role Instruction (Ours)	**64.4**	**51.9**

[I] We sample 4 HumanEval tasks to construct examples for few-shot prompting, which are excluded in evaluations of 'The Role of Role-playing'.

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.

Second, we find that the performance of 'Analyst + Coder + Tester + Compiler' is comparable to that of 'Analyst + Coder + Tester'. The reason may be that some mistakes can already be resolved by the tester without relying on the complier's results. Moreover, the team of pair programming is superior to the single LLM agent, but still performed slightly worse compared to the team of 'Analyst + Coder + Tester'.

Third, the results 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. 1) Few-shot prompting may bias the LLMs' understanding of human intent due to the limited selection of examples that might not fully reflect intent. 2) 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.

4.4 RQ3: Self-collaboration on Different LLMs

In addition to ChatGPT, we apply self-collaboration to seven open-source LLMs, including Fastchat-3B [32], ChatGLM-6B [54], MPT-7B [36], Vicuna-7B/13B [33], HuggingChat-30B [21], and Dromedary-65B [22], and the most powerful LLM GPT-4 [40]. Our objective is to evaluate the efficacy of self-collaboration in tackling complex tasks, specifically those challenging for the single LLM agent. For such tasks on HumanEval, we employ the self-collaboration strategy as the solution. As illustrated in Fig. 4, 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

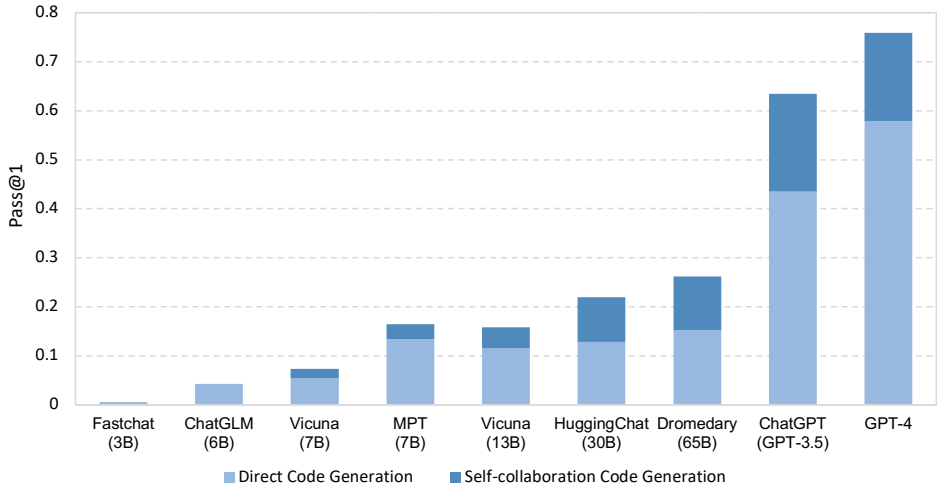

Fig. 4. Self-collaboration capacities of different LLMs.

7B parameters and subsequently continues to escalate, serving to evoke latent intelligence within LLMs.

For the most powerful LLM GPT-4, we conduct additional experiments to evaluate self-collaboration code generation based on GPT-4, following the settings in the GPT-4 technical report [40]. The experimental results are shown in Table 5, and we can find that the enhancement effect of self-collaboration on GPT-4 is significant.

Table 5. The performance of self-collaboration code generation with GPT-4. The result in brackets is reported on the GPT-4 technical report [40].

Approach	HumanEval	HumanEval-ET	MBPP	MBPP-ET
GPT-4 [40]	67.7 (67.0)	50.6	68.1	49.2
GPT-4 + Self-collaboration	90.2	70.7	78.9	62.1

To figure out the abilities required for self-collaboration, we conduct experiments on a series of models including CodeLlama 7B and 34B [45], Llama2 [55] (the base model of CodeLlama) and their Instruct version (model with instruction tuning). As illustrated in Figure 5, the enhancement observed in Llama2-7B using self-collaboration falls short when compared to CodeLlama-7B. This discrepancy emphasizes the critical role of domain-specific expertise. The performance improvement of the Instruct version using self-collaboration generally exceeds that of the original version, highlighting the significance of in-context learning capabilities. Furthermore, the advancement in CodeLlama-34B using self-collaboration eclipses that of CodeLlama-7B on both two versions, underscoring the importance of reasoning abilities. Therefore, self-collaboration may require the following abilities for LLMs, including strong domain-specific expertise ability for role-playing, strong in-context learning ability to follow instructions, and strong reasoning ability to solve problems effectively.

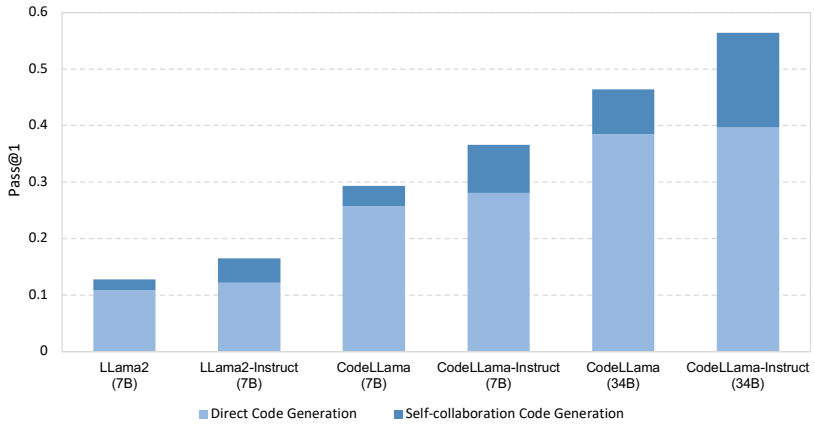

Fig. 5. Self-collaboration capacities of CodeLlama series.

4.5 RQ4: The Effect of Interaction

To evaluate the impact of interaction on self-collaboration code generation, we control the maximum number of interactions (MI) between roles in this experiment, with the results shown in Table 6. 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 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 most significant 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 maximum token constraint of ChatGPT, our exploration was limited to the MI value from 1 to 4 rounds.

Table 6. 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

4.6 RQ5: Analysis for Self-collaboration

In this section, we further analyze the errors generated by self-collaboration and the cost of self-collaboration compared to other prompting approaches.

4.6.1 Error Analysis. To deepen the understanding of error modes for self-collaboration, we conducted a qualitative error analysis. On HumanEval dataset, we classify tasks into three categories: the first category includes tasks that can be correctly generated by the coder; the second category

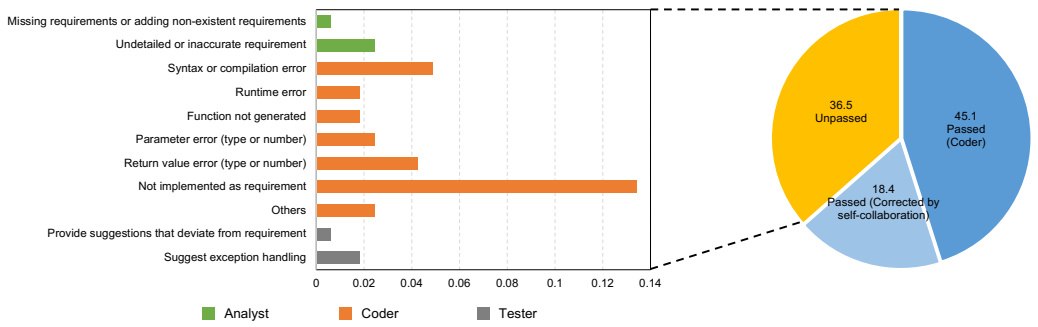

Fig. 6. Error Analysis for Self-collaboration.

includes tasks that cannot be correctly generated by the coder itself but can be generated correctly through introducing self-collaboration; and the third category includes tasks that cannot be correctly generated even with self-collaboration. By manually checking the tasks in the third category, we attribute the responsibility for generating errors to different roles and further subdivide the types of errors in Figure 6. From the findings, it is evident that the predominant error originates from the coder. This error is inherently tied to the performance of the base model. However, the introduction of self-collaboration markedly improves the quality of code generation, where 18.4% of tasks unpassed by coder are corrected by self-collaboration. Moreover, the errors caused by the analyst mainly stem from ambiguous or incomplete requirement descriptions, which cannot be solely attributed to the analyst. Remedying this issue can be alleviated by providing more precise and complete requirement descriptions or incorporating a small amount of human guidance. On the other hand, errors associated with testers predominantly result from exception handling. These adjustments usually do not introduce new errors and tend to enhance the safety of the program. However, they may cause some instances to throw an error instead of returning a value, thus failing the test case.

In addition to the preceding problems, the probability of deviating from the requirements caused by the additional introduction of the analyst and tester is less than 1%, so the whole system is relatively reliable.

4.6.2 Cost Analysis. As illustrated in Figure 7, we measure the cost (prompt tokens + generated tokens) and performance of self-collaboration, coder, and other prompting approaches, where we normalize the cost of coder as 1. The experimental results indicate that the improvement of the self-collaboration approach is significant, and its token usage is moderate among all prompting approaches. However, considering the high labor expenses of software development teams, the cost-effectiveness of self-collaboration is obvious.

4.7 RQ7: 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, which aims to demonstrate the workflow of our virtual team. The second involves a more complex real-world requirement, specifically asking model to create a project in Python, which is a high-level requirement similar to a user story. The extended case study can be found in Appendix D.

Fig. 7. Performance and Cost of Self-collaboration Compared to Baselines.

4.7.1 Workflow. We illustrate the workflow and performance of self-collaboration on HumaneEval benchmark. A case study is presented in Fig. 8, which exemplifies how the three roles work collaboratively to solve code generation in self-collaboration. 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, 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.

4.7.2 Complex Task. We apply self-collaboration code generation to a complex game development requirement, with a case study shown in Fig. 9. 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.

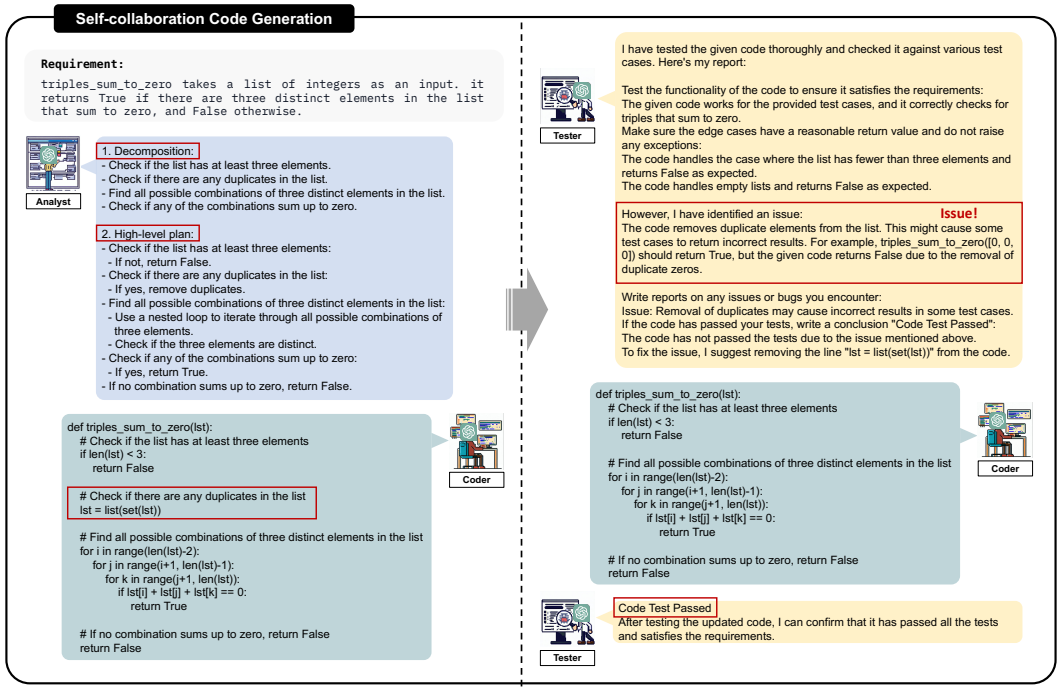

Fig. 8. Case study on HumanEval benchmark.

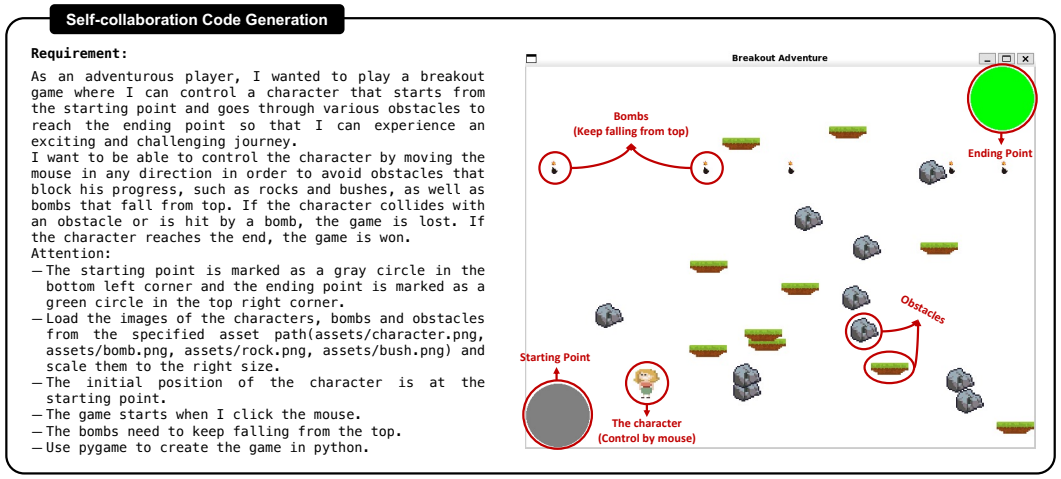

Fig. 9. Case study on complex tasks in real-world scenarios. Red markers are added to denote specific objects.

We also 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. The case study of the website development is shown in Fig. 10. Our self-collaboration code generation approach produces a website that is superior to ChatGPT direct generation approach in terms of

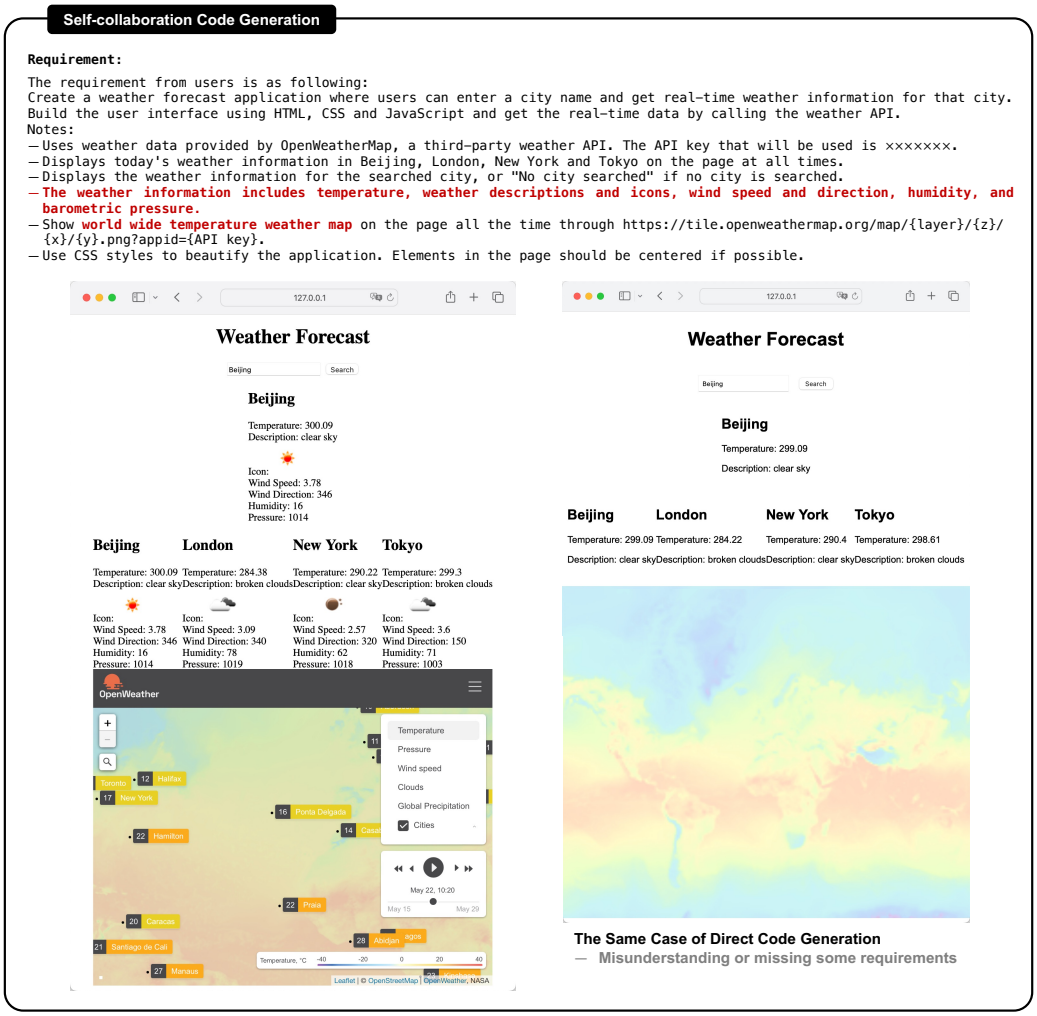

Fig. 10. Case study on complex tasks in real-world scenarios.

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.

5 RELATED WORK

In this section, we outline the most relevant directions and associated papers of this work to highlight our research's innovative and advanced nature.

5.1 Multi-agent Collaberation

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 [52]. This direction has been under exploration for a considerable length of time [12, 35]. 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 [58] and HuggingGPT [50] 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 [28] 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 and significant 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. To the best of our knowledge, we are the first to introduce the waterfall model in software-development methodology to the collaboration among LLMs. We demonstrate the feasibility of this approach and provide the direction for future work on multi-agent collaboration.

5.2 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 [6, 17, 31].

Recent studies have shown that well-designed prompting methods, such as chain-of-thought prompting (CoT) [56] and zero-shot CoT [27], 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 enhance 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 [24, 44, 51, 65]. 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 to enable the same LLM to be differentiated into different roles, and demonstrate the effectiveness of role-playing. 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.

5.3 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 [2] explores the potential of LLMs in driving requirements engineering processes. The work [62] applied LLMs to code generation and demonstrated significant improvement in the pass rate of generated complex programs. The work [25] employed LLMs for generating tests

to reproduce a given bug report and found that this approach holds great potential in enhancing developer efficiency. The work [47] also demonstrates the effectiveness of applying LLMs to test generation. In a comprehensive study conducted by the work [59], 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 [60] 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.

5.4 Code Generation with Other Stage of Software Development

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. CodeT [7] employs a language model to automatically generate test cases for code samples and subsequently utilizes a dual execution agreement to rank the code. Coder-Reviewer [63] introduces an additional reviewer model to enhance the coder model, which generates requirements based on the code and calculates the maximum likelihood to rank the generated code. They provide no feedback to the model, so that falls under the category of post-processing. Recently, another type of approach focuses on improving the quality of generated code. Self-planning [23] introduces a planning stage prior to code generation. Self-debugging [9] teaches LLM to perform rubber duck debugging after code generation. They rely on few-shot prompting to instruct LLM, which requires a certain amount of effort to construct demonstration examples as prompts 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 as an 'expert' responsible for a distinct stage and provides feedback to each other to improve the quality of the final generated code. Moreover, we introduce software-development methodology, which is imperative for solving complex code generation tasks and practical software development problems.

6 DISCUSSION AND FUTURE WORK

In this section, we discuss some limitations of this work and present a set of potential directions for future research.

1) There are still some gaps between the evaluation benchmark we used and the actual software development scenarios. Because there is still a lack of benchmarks and corresponding evaluation metrics that can truly reflect actual software development tasks. In order to alleviate this problem, we have selected some more complex tasks originating from actual scenarios for the case study.

2) Although the virtual team we have assembled is a fully autonomous system and the roles in the virtual team can monitor and correct each other, incorporating a small amount of guidance from human experts to oversee the functioning of the virtual team would help enhance the utility of our approach in real-world application scenarios.

3) In this study, we constructed a virtual development team consisting of three roles to target the code generation task. However, for other problems, the team structure may need to be adjusted accordingly. It is worth emphasizing that our proposed approach is not limited to the traditional software engineering domain, and it has the potential to build virtual teams consisting of specific roles for a variety of problems in other domains as well.

7 CONCLUSION

In this paper, we have proposed a self-collaboration framework designed to enhance the problem-solving capability of LLMs in a collaborative and interactive way. We investigate the potential of LLMs in facilitating collaborative code generation within software development processes. Specifically, based on the proposed framework, we assemble an elementary team consisting of three distinct LLM agents, designed to address code generation tasks collaboratively. Extensive experimental results demonstrate the effectiveness and generalizability of self-collaboration framework. In conclusion, self-collaboration framework provides an effective approach to automatic code generation. This innovative approach has the potential to substantially improve the quality of generated code, reduce human intervention, and accelerate the development of complex software systems. Moreover, our work can serve as a foundation for future research on multi-agent collaboration approaches in various domains and the development of more advanced and specialized virtual teams to tackle more complex tasks.

REFERENCES

[1] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. 2002. Agile software development methods: Review and analysis. (2002).

[2] Chetan Arora, John Grundy, and Mohamed Abdelrazek. 2023. Advancing Requirements Engineering through Generative AI: Assessing the Role of LLMs. *CoRR* abs/2310.13976 (2023).

[3] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021).

[4] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. 2001. Manifesto for agile software development. (2001).

[5] R Meredith Belbin. 2012. *Team roles at work*. Routledge.

[6] 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. 2020. Language Models are Few-Shot Learners. In *NeurIPS 2020*.

[7] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT: Code Generation with Generated Tests. *CoRR* abs/2207.10397 (2022).

[8] 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 Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* (2021). https://arxiv.org/abs/2107.03374

[9] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. *CoRR* abs/2304.05128 (2023).

[10] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways. *CoRR* abs/2204.02311 (2022).

[11] 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. 2022. Scaling Instruction-Finetuned Language Models. *CoRR* abs/2210.11416 (2022).
[12] Caroline Claus and Craig Boutilier. 1998. The Dynamics of Reinforcement Learning in Cooperative Multiagent Systems. In *AAAI/IAAI*. AAAI Press / The MIT Press, 746–752.
[13] Tom DeMarco and Tim Lister. 2013. *Peopleware: productive projects and teams*. Addison-Wesley.
[14] Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. 2023. CodeScore: Evaluating Code Generation by Learning Code Execution. *CoRR* abs/2301.09043 (2023).
[15] Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, and Ge Li. 2024. Generalization or Memorization: Data Contamination and Trustworthy Evaluation for Large Language Models. *CoRR* abs/2402.15938 (2024).
[16] Yihong Dong, Ge Li, and Zhi Jin. 2023. CODEP: Grammatical Seq2Seq Model for General-Purpose Code Generation. In *ISSTA*. ACM, 188–198.
[17] Yihong Dong, Kangcheng Luo, Xue Jiang, Zhi Jin, and Ge Li. 2023. PACE: Improving Prompt with Actor-Critic Editing for Large Language Model. *CoRR* abs/2308.10088 (2023).
[18] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. *CoRR* abs/2204.05999 (2022).
[19] GitHub. 2022. *Copilot*. https://github.com/features/copilot
[20] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. In *NeurIPS Datasets and Benchmarks*.
[21] Huggingface. 2023. *HuggingChat*. https://huggingface.co/chat/
[22] IBM. 2023. *Dromedary*. https://huggingface.co/ibm/dromedary-65b-lora-delta-v0
[23] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning Code Generation with Large Language Model. *CoRR* abs/2303.06689 (2023).
[24] Zhengbao Jiang, Frank F. Xu, Jun Araki, and Graham Neubig. 2020. How Can We Know What Language Models Know. *Trans. Assoc. Comput. Linguistics* 8 (2020), 423–438.
[25] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2022. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. *CoRR* abs/2209.11515 (2022).
[26] Jon R Katzenbach and Douglas K Smith. 2015. *The wisdom of teams: Creating the high-performance organization*. Harvard Business Review Press.
[27] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models are Zero-Shot Reasoners. In *NeurIPS*.
[28] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. CAMEL: Communicative Agents for" Mind" Exploration of Large Scale Language Model Society. *CoRR* abs/2303.17760 (2023).
[29] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *CoRR* abs/2305.06161 (2023).
[30] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.
[31] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9 (2023), 195:1–195:35.
[32] LMSYS. 2023. *Fastchat*. https://huggingface.co/lmsys/fastchat-t5-3b-v1.0
[33] LMSYS. 2023. *Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality*. https://lmsys.org/blog/2023-03-30-vicuna/

[34] Ian R McChesney and Seamus Gallagher. 2004. Communication and co-ordination practices in software engineering projects. *Information and Software Technology* 46, 7 (2004), 473–489.
[35] Marvin Minsky. 2007. *The emotion machine: Commonsense thinking, artificial intelligence, and the future of the human mind.* Simon and Schuster.
[36] MosaicML. 2023. *Introducing MPT-7B: A New Standard for Open-Source, ly Usable LLMs.* www.mosaicml.com/blog/mpt-7b
[37] H Penny Nii. 1986. Blackboard Systems. (1986).
[38] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *CoRR* abs/2203.13474 (2022).
[39] OpenAI. 2022. *ChatGPT.* https://openai.com/blog/chatgpt/
[40] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023).
[41] 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. 2022. Training language models to follow instructions with human feedback. *CoRR* abs/2203.02155 (2022).
[42] 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. 2022. Training language models to follow instructions with human feedback. *CoRR* abs/2203.02155 (2022).
[43] Kai Petersen, Claes Wohlin, and Dejan Baca. 2009. The Waterfall Model in Large-Scale Development. In *PROFES (Lecture years in Business Information Processing, Vol. 32)*. Springer, 386–400.
[44] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. In *CHI Extended Abstracts*. ACM, 314:1–314:7.
[45] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023).
[46] Nayan B. Ruparelia. 2010. Software development lifecycle models. *ACM SIGSOFT Softw. Eng. years* 35, 3 (2010), 8–13.
[47] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Trans. Software Eng.* 50, 1 (2024), 85–105.
[48] Timo Schick, Jane Dwivedi-Yu, Zhengbao Jiang, Fabio Petroni, Patrick S. H. Lewis, Gautier Izacard, Qingfei You, Christoforos Nalmpantis, Edouard Grave, and Sebastian Riedel. 2022. PEER: A Collaborative Language Model. *CoRR* abs/2208.11663 (2022).
[49] Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li. 2022. Incorporating domain knowledge through task augmentation for front-end JavaScript code generation. In *ESEC/SIGSOFT FSE*. ACM, 1533–1543.
[50] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *CoRR* abs/2303.17580 (2023).
[51] Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. 2020. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In *EMNLP (1)*. Association for Computational Linguistics, 4222–4235.
[52] Stephen W. Smoliar. 1991. Marvin Minsky, The Society of Mind. *Artif. Intell.* 48, 3 (1991), 349–370.
[53] Chang-You Tai, Ziru Chen, Tianshu Zhang, Xiang Deng, and Huan Sun. 2023. Exploring Chain-of-Thought Style Prompting for Text-to-SQL. *CoRR* abs/2305.14215 (2023).
[54] THUDM. 2023. *ChatGLM.* https://huggingface.co/THUDM/chatglm-6b
[55] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023).
[56] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *CoRR* abs/2201.11903 (2022).

[57] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*.

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

[59] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical Program Repair in the Era of Large Pre-trained Language Models. *CoRR* abs/2210.14179 (2022).

[60] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. *CoRR* abs/2301.13246 (2023).

[61] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *ICSE*. ACM, 37:1–37:12.

[62] Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D. Goodman, and Nick Haber. 2022. Parsel: A Unified Natural Language Framework for Algorithmic Reasoning. *CoRR* abs/2212.10561 (2022).

[63] Tianyi Zhang, Tao Yu, Tatsunori B. Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida I. Wang. 2022. Coder Reviewer Reranking for Code Generation. *CoRR* abs/2211.16490 (2022).

[64] 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. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. *CoRR* abs/2303.17568.

[65] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large Language Models Are Human-Level Prompt Engineers. *CoRR* abs/2211.01910 (2022).

A PRELIMINARY KNOWLEDGE

A.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.

A.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.

A.3 Waterfall Model

The waterfall development model is the most classic software development methodology, which has the following advantages: 1) Its linear and sequential nature facilitates ease of understanding and implementation, particularly for LLMs. 2) Each phase within this model is distinctly defined, accompanied by specific deliverables and a rigorous review process, which enhances manageability. 3) It enables the early detection and rectification of issues, which can significantly mitigate the costs and adverse impacts of errors identified in subsequent phases of the project. For alternative approaches, one may consider pair programming, test-driven development, and other software development methodologies.

B DETAILED SETTINGS AND BASELINES

In this paper, we employ two prevalent settings for code generation: The first setting, referred to as NL + signature + use cases, provides an NL description, function signature, and use 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, names, and return value type, as do use cases.
(2) This setting is mainly suited for function-level code generation and proves challenging to extend to file-level or repository-level code generation.
(3) Some code-generation benchmarks, such as MBPP, do not provide function signatures and use 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.

C ABLATION STUDY WITH ROLES IN SELF-COLLABORATION CODE GENERATION.

We verify the performance of roles in self-collaboration code generation in 'NL + signature + use cases' settings where signatures and use cases are provided in addition to NL. The results of this evaluation are displayed in Table 7. As demonstrated in the findings, the self-collaboration code generation is far more effective than the single LLM agent, 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 7. Effectiveness of ChatGPT roles in self-collaboration code generation with 'NL + signature + use cases' setting.

Approach	HumanEval	HumanEval-ET	MBPP	MBPP-ET
Direct	57.3	42.7	52.2	36.8
Self-collaboration (Virtual Team)				
Analyst-Coder	73.5 (↑ 28.2%)	55.2 (↑ 29.2%)	65.3 (↑ 25.1%)	48.9 (↑ 32.9%)
Coder-Tester	60.3 (↑ 5.3%)	45.2 (↑ 5.9%)	64.2 (↑ 23.0%)	48.3 (↑ 31.3%)
Analyst-Coder-Tester	**74.4** (↑ 29.9%)	**56.1** (↑ 31.4%)	**68.2** (↑ 30.7%)	**49.5** (↑ 34.6%)

We also performed ablation studies with roles on MBPP in 'NL-only' setting. benchmark. The results are shown in Table 8. From the results, it is clear that different roles perform differently on MBPP than on HumanEval. The coder-tester team attained the highest performance on MBPP and MBPP-ET benchmarks, with relative improvements of 36.7% and 39.4%, respectively.

Table 8. Effectiveness of ChatGPT roles in self-collaboration code generation on MBPP with 'NL-only' Setting.

Roles	MBPP	MBPP-ET
Coder	47.5	34.3
+ Analyst	54.8 (↑ 15.4%)	39.9 (↑ 16.4%)
+ Tester	**64.9** (↑ 36.7%)	**47.8** (↑ 39.4%)
+ Analyst + Tester	55.5 (↑ 16.9%)	40.8 (↑ 19.0%)

The suboptimal performance of the analyst on MBPP and MBPP-ET benchmarks may be attributed to 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 requirement. Analysis on such unconventional data usually yields results that align with human habits, while coding directly based on requirements can handle this unconventional data. Based on our experimental results and the phenomenon we observed, we suspect there might be some degree of data contamination in MBPP. However, considering this fact, we still choose to publish these results. Under 'NL + signature + use cases' setting, LLM has access to more comprehensive input-output information, and we find that the role of the analyst is significant.

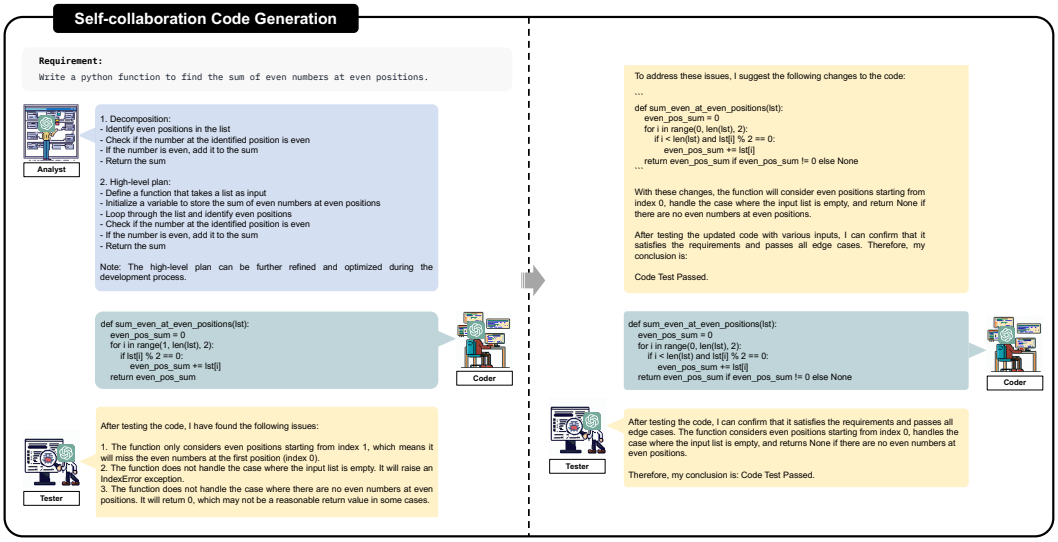

Fig. 11. Case study on MBPP benchmark.

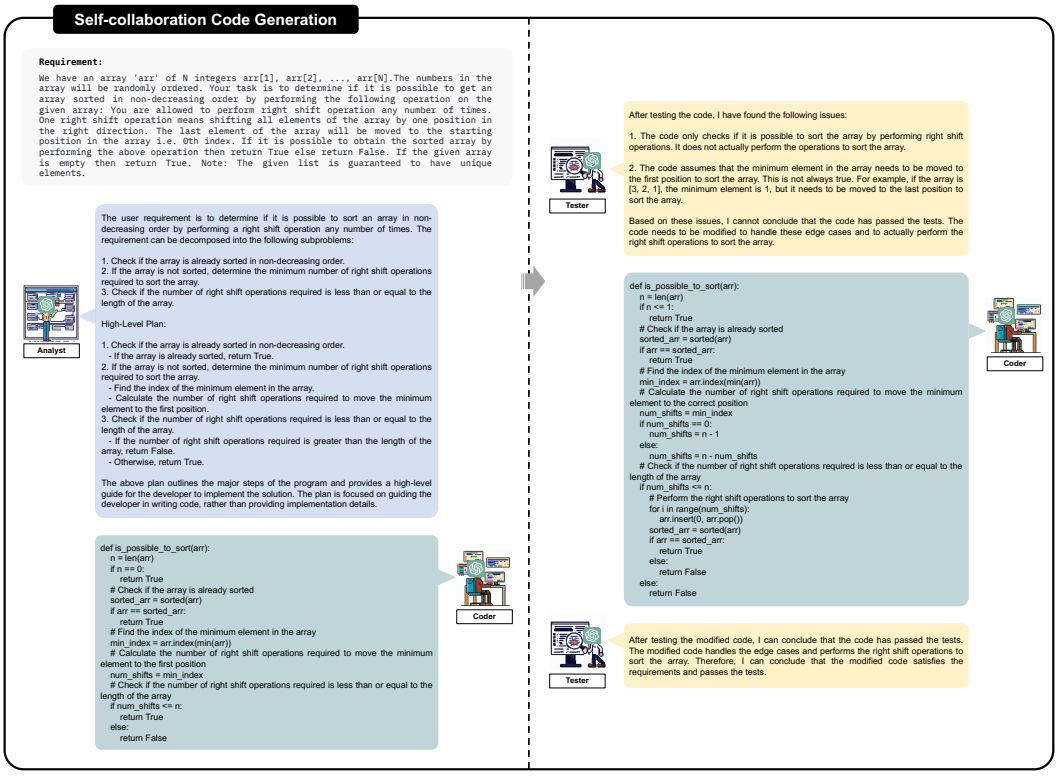

Fig. 12. Case study on HumanEval benchmark.

D EXTENDED CASE STUDY

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. 11 and Fig. 12. These examples illustrate the efficacy of our self-collaboration code generation approach in addressing challenges that are difficult to overcome using the single LLM agent.

Setup. We employ GPT-4 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.

E THE INSTRUCTION OF BASELINES PAIR PROGRAMMING

We configure a type of pair programming team whose role instructions are as follows.

DRIVER = "I want you to act as the driver in this team. Your job is as follows:
1. You are responsible for writing code, i.e. translating your understanding of the requirement into code.
2. You need to explain the code to help the observer understand what you have written.
3. If you receive suggestions from the observer, you need to fix or improve your code based on his suggestions. Ensure that any changes made to the code do not introduce new bugs or negatively impact the performance of the code."

OBSERVER = "I want you to act as the observer in this team. You will receive the code written by the driver, and your job is as follows:
1. You are primarily responsible for reviewing code written by drivers to ensure its quality and accuracy. You need to provide suggestions on the code written by the drivers.
2. You also need to think about the needs that the code meets.
3. You also need to predict possible problems and errors and instruct drivers to correct them."

TEAM = "There is a pair programming team that includes a driver, and an observer. The team needs to develop a program that meets a requirement. The different roles have different divisions of labor and need to cooperate with each others."

F 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 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 indicies, and equal to the sorted even indices in the even indicies.
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 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".
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 indicies, while its values at the even indicies are equal to the values of the even indicies 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 indicies, while its values at the even indicies are equal to the values of the even indicies 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 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']
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: ”