arXiv:2312.13010v2 [cs.CL] 23 Jan 2024

# AgentCoder: Multiagent-Code Generation with Iterative Testing and Optimisation

**Dong Huang**[1], **Qingwen Bu**[2], **Jie M. Zhang**[3], **Michael Luck**[4] and **Heming Cui**[1]

[1]The University of Hong Kong
[2]Shanghai Jiao Tong University
[3] King's College London
[4] University of Sussex
{dhuang, heming}@cs.hku.hk, qwbu01@sjtu.edu.cn, jie.zhang@kcl.ac.uk, Michael.Luck@sussex.ac.uk

## Abstract

Advances in natural language processing (NLP) have been significantly boosted by the development of transformer-based large language models (LLMs). These models have revolutionized NLP tasks, particularly in code generation, aiding developers in creating software with enhanced efficiency. Despite their advances, challenges remain in balancing code snippet generation with effective test case generation and execution. To address these issues, this paper introduces Multiagent-Code Generation (AgentCoder), a novel solution comprising a multi-agent framework with specialized agents: the programmer agent, the test designer agent, and the test executor agent. During the coding procedure, the programmer agent focuses on the code generation and refinement based on the test executor agent's feedback. The test designer agent generates test cases for the generated code, and the test executor agent runs the code with the test cases and writes feedback to the programmer. This collaborative system ensures more effective code generation, surpassing the limitations of single-agent models and previous strategies. Our extensive experiments on 12 LLMs and 13 optimisation approaches showcase AgentCoder's superior performance over existing code generation models and prompt engineering techniques across various benchmarks. For example, AgentCoder achieves 77.4% and 89.1% pass@1 in HumanEval-ET and MBPP-ET with GPT-3.5, while state-of-the-art obtains only 69.5% and 63.0%.

## 1 Introduction

In recent years, natural language processing (NLP) has been dramatically transformed by transformer-based large language models (LLMs). These models, notably exemplified by the GPT-x series [Brown *et al.*, 2020b; OpenAI, 2023] developed by OpenAI, have consistently set the benchmark for performance across a wide array of standard NLP tasks. One of the most pivotal applications for these LLMs is code generation for downstream tasks, where they play a vital role in aiding developers in creating software [Feng *et al.*, 2020; Wang *et al.*, 2021; Wang *et al.*, 2023b; Nijkamp *et al.*, 2023b; Nijkamp *et al.*, 2023a; Li *et al.*, 2023b]. Through extensive pretraining on substantial code-related datasets, such as publicly available data on GitHub, these code LLMs acquire intricate contextual understanding that can be effectively applied to diverse code-related tasks.

Numerous recent efforts have been made to improve the effectiveness of code generation models by incorporating in-context learning and its variations [Dong *et al.*, 2023b; Wei *et al.*, 2022; Le *et al.*, 2023; Huang *et al.*, 2023; Zhang *et al.*, 2023b; Chen *et al.*, 2023b; Madaan *et al.*, 2023], where an important optimization path is self-refinement. For example, Zhang *et al.* proposed Self-Edit to enhance the performance of LLMs in code generation. In particular, Self-Edit runs the code generation model's generated code against test cases that are manually written by developers. If the code fails to pass these test cases, Self-Edit prompts the code generation model to refine the function using the provided error messages with its fault-aware code editor. Nevertheless, Self-Edit requires that developers write test cases to verify the correctness of the generated function. This requirement can be particularly demanding and challenging for users who lack expertise in the specific domain, which potentially impedes the effectiveness of the self-editing process.

To overcome this challenge, Huang *et al.* introduced CodeCoT, which adopts a step-by-step strategy for code generation, tasking the code generation model to generate both the function and the corresponding test cases. CodeCoT also establishes a connection with a terminal interface, instructing the code generation model to self-refine the code based on the error messages returned by the terminal. This approach not only reduces the burden on developers in terms of writing test cases but also ensures that the generated code undergoes software testing and refinement.

Although CodeCoT makes substantial strides in enhancing the effectiveness of code generation models, the tests and code are generated within the same conversation. In other words, the code generation and test generation processes are not independent. This practice brings constraints that arise from the potential trade-off between excelling in code generation and maintaining the effectiveness of test case generation: as the model achieves high performance in generating code snippets, there may be a corresponding decrease in the effectiveness of test case generation [Chen *et al.*, 2023a; Zhang *et al.*, 2023a]. This trade-off scenario occurs due to the model's limited resources and its focus on optimizing one aspect of the code generation process, which might inadvertently compromise the quality of other tasks [Chen *et al.*, 2023a; Zhang *et al.*, 2023a]. In addition, the tests generated immediately following the code in one conversation can be biased and affected by the code, losing objectivity and diversity in the testing (See Tab. 5).

In this paper, we address the above-mentioned problems by proposing Multiagent-Code Generation, namely AgentCoder. AgentCoder contains three different agents, i.e., the programmer agent, the test designer agent, and the test executor agent. The programmer agent interacts with advanced code generation models to create code based on coding requirements. The test designer agent designs diverse and comprehensive test cases with code generation models independently based on the coding requirements. The test executor agent interacts with both the programmer agent and the test designer agent: it executes the tests from the test designer agent against the code generated by the programmer agent and then provides test execution results to the programmer agent. Once the feedback is obtained by the test executor agent from the local environment (i.e., local terminal), it checks whether the feedback contains error information (e.g., runtime error and assertion error). If all test cases pass the generated code, the test executor agent provides the code snippets with the human developer. Otherwise, the test executor agent feeds back to the programmer agent and then requires it to fix the bug reported in the feedback. Then the iteration continues once the feedback is that all test cases pass the code snippets or the iteration budget is done, when the code snippets will be reported to the human developer even if the code is still buggy.

Our extensive experiments with 12 LLMs and 13 enhancement approaches demonstrate that AgentCoder significantly improves the effectiveness of existing code generation models, outperforming all baseline approaches. In particular, AgentCoder obtains an average of 91.5% and 84.1% pass@1 on all the datasets with GPT-4 and GPT-3.5, respectively, while the state-of-the-art approaches obtain 86.8% and 75.3%. On HumanEval-ET and MBPP-ET, AgentCoder obtains 77.4% and 89.1% pass@1 with GPT-3.5, while the state-of-the-art approaches obtain only 69.5% and 63.0%. The effectiveness of AgentCoder is fueled by the goal of leveraging collaborative synergy within its agents. Within this agent system, the programmer agent excels in crafting high-quality code snippets, complemented by the test designer agent's expertise in designing varied, challenging, and objective test cases. The test executor agent plays a pivotal role by critically evaluating the code using these test cases, ensuring both functionality and reliability. Such collaboration fosters a dynamic feedback loop that facilitates successive enhancements. AgentCoder overcomes the constraints inherent in single-agent code generation models by allocating distinct tasks to different agents. This division not only balances the focus between code and test case generation but also strengthens a more objective testing process. Additionally, its modular design provides the flexibility and scalability crucial to adapting to technological advancements. Agents within AgentCoder can be individually updated or replaced with more sophisticated models, maintaining the framework's technological edge. This adaptability positions AgentCoder as an effective and evolving solution in the ever-changing landscape of software development.

Our main contributions are as follows:

- Introduction of AgentCoder: We propose AgentCoder, a novel multi-agent framework for code generation that contains three distinct agents, i.e., the programmer agent, the test designer agent, and the test executor agent.
- Comprehensive Evaluation: We conduct an extensive evaluation with 12 LLMs and 13 LLM-based optimisation approaches which demonstrates that AgentCoder outperforms all the baselines in code generation. In particular, AgentCoder obtains 77.4% and 89.1% pass@1 with GPT-3.5, while state-of-the-art obtains only 69.5% and 63.0%.
- In-Depth Analysis and Ablation Studies: We conduct a deep analysis of our results and ablation studies, which demonstrate the contribution of different agents, the effectiveness of the tests generated by the test designer agent, and the necessity of using separate agents for code generation and test case design.
- Modularity: The modular structure of our framework not only ensures adaptability and scalability but also facilitates future enhancements and integration with more advanced models, positioning AgentCoder as a resilient solution in the evolving landscape of code generation.

## 2 Related Work

### 2.1 Large Language Model for Code Generation

Large Language Models (LLMs) have been widely studied for code generation tasks. Various architectures have been explored in these models, some notable examples being CodeBERT [Feng *et al.*, 2020], PLBART [Ahmad *et al.*, 2021], and CodeGPT [Zan *et al.*, 2022]. These models are pretrained on code corpora to develop a deep understanding of code syntax, semantics, and idiomatic constructs. Some innovative approaches integrate structured representations to enhance their comprehension of the complexities in code. For example, GraphCodeBERT [Guo *et al.*, 2020] incorporates graph-based representations, while CodeT5+ [Wang *et al.*, 2023b] combines the encoder-decoder paradigm with the structural essence of code. These enhancements aim to give the models a more fine-grained understanding of code relationships and dependencies beyond just syntactic patterns. A current trend is the construction of large scale models (e.g., Codex [Chen *et al.*, 2021b] and CodeGen [Nijkamp *et al.*, 2023b]) with billions of parameters, which have illustrated the performance of state-of-the-art in code generation tasks. Recently, foundation models (e.g., GPT-3.5-turbo, GPT-4) have also been used for code generations [Madaan *et al.*, 2023; Huang *et al.*, 2023]. These foundation models illustrated the state-of-the-art performance for code generation tasks.

### 2.2 Enhancing Code Generation through Prompt Engineering

Recent advances in code generation have been significantly influenced by the integration of few-shot learning techniques with LLMs. A notable contribution in this realm is the concept of self-refinement with few-shot prompting, as proposed by Madaan *et al.*. This approach involves an LLM iteratively refining its own generated code, leading to significant improvement in code quality. Another approach is the Self-Debugging technique introduced by Chen *et al.*, which involves testing the generated code against user-provided test cases. In scenarios where such test cases are unavailable, the model engages in direct debugging by explaining the code, thus addressing potential issues. Complementing these methods, Huang *et al.* introduced CodeCoT, employing a Self-Exam Chain of Thought (CoT) process. This technique guides the model to generate code alongside test cases, particularly useful when external test cases are not available. CodeCoT adds a layer of logical reasoning to the code generation process. However, it is important to note that while this method can identify syntax errors, functional errors may still go undetected as both the code and its test cases are generated by the same model. Building upon these concepts, Dong *et al.* proposed the Self-Collaboration model, which divides the LLMs into different roles: an analyst, a coder, and a tester. The tester is powered by an LLM which predicts whether the code is buggy. Such practice may ignore many bugs in the code because the code is not executed in the local environments.

### 2.3 Multi-agent Collaboration

A multi-agent system (MAS) is a framework where multiple autonomous agents interact with each other. These agents, which can be program scripts, software bots, or robots, operate in a shared environment and can communicate, cooperate, compete, or negotiate with each other. Each agent in a multi-agent system has its own capabilities, goals, and perceptions, and works either independently or together to achieve complex goals or solve problems. The integration of LLMs within multi-agent collaboration systems represents a cutting-edge area of research in the deep learning community. For example, HuggingFace proposes HuggingGPT to solve complex AI tasks with HuggingFace models. Zhang *et al.* propose ProAgent to address robotic tasks by analyzing the current context, anticipating teammates' intentions, and formulating its strategies based on the above reasoning. Chen *et al.* propose VisualGPT to utilize vision PLM to address image captioning tasks.

## 3 Methodology

The framework of AgentCoder and its pipeline are illustrated in Fig. 1. The process begins by inputting tasks/code generation requirements/descriptions into the code generation agent (Agent#1: the programmer agent). Subsequently, the test case generator (Agent#2: the test designer agent) is tasked with generating test cases, which are used to evaluate the correctness of the code snippets produced by the programmer agent. The code snippets and test cases are collected by the test executor agent (Agent#3) and executed in the local environment (local terminal) to obtain feedback (i.e., whether the code passes all tests and the error message if the code fails for some tests). If the test executor agent finds that the code snippets pass all test cases, it will return the code to the user and finish the iteration. Otherwise, the test executor agent will return the test execution error messages to the programmer agent. The iteration then continues, with the programmer agent regenerating code snippets to address the issues identified in the feedback, and the test executor agent re-executes the new code and provides new feedback to the programmer agent, until the test executor agent finds that the code passes all the tests.

### 3.1 Programmer agent: code generation with Chain-of-Thought instruction

In our framework, The programmer agent is powered by LLMs. It needs to consider two scenarios, i.e., code generation and code refinement. Specifically, as shown in Fig. 1, during the code generation stage, the human developer will require the programmer agent to generate code snippets to complete specific tasks, the programmer agent employs a Chain-of-Thought approach to simulate the typical programming process, methodically breaking down the task into smaller, manageable steps. The Chain-of-Thought process is instructed to contain four steps, i.e., problem understanding and clarification, algorithm and method selection, pseudocode creation, and code generation (the prompt and response example is shown in Appendix A.3 Figure 6 and 7).

Taking the coding task *Check if in given list of numbers, are any two numbers closer to each other than given threshold* (shown in Figure 1) as an example, during the initial code generation, the programmer agent will try to understand and clarify the given task, in this case interpreting the requirement to identify pairs of numbers in a list that are within a specified threshold of each other. The programmer agent will then decide on an algorithm or method to solve the problem. This could involve choosing an efficient way to compare each pair of numbers in the list. Next, during the pseudocode creation, the programmer agent will develop a step-by-step guide or pseudocode for the solution, ensuring a logical flow of operations. Finally, in the code generation stage, the programmer will translate the pseudocode into executable code.

Code snippets generated by the programmer agent can be incorrect, containing various types of errors (e.g., syntax and runtime errors), leading to failed test cases provided by the test designer agent. Under such circumstances, the programmer agent will take feedback from other agents and refine the code snippets. The refinement process is iterative, with the programmer agent continuously enhancing the code based on feedback until the code successfully passes all test cases.

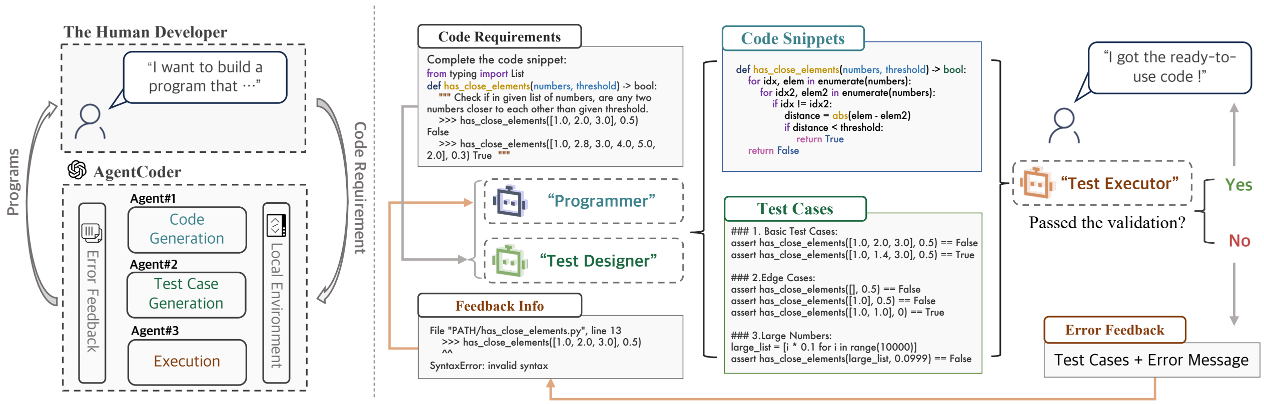


Figure 1: Pipeline of AgentCoder with a code generation example from HumanEval

### 3.2 Test designer agent: generating basic, edge, and large scale tests

The test designer agent is also powered by LLMs. It is a crucial component of our AgentCoder's framework to test the code and provide reliable feedback for the programmer agent to optimise the code iteratively. We carefully designed the prompts for the test designer agent to satisfy the following three expectations: (i) to generate basic test cases, (ii) to cover edge test cases, and (iii) to cover large scale inputs (the test designer agent's prompt and response example is shown in Appendix Figure 8 and 9). The first aspect expects that the test designer agent designs test cases that cover the fundamental functionality of the code. These tests are designed to ensure that the code performs as expected under normal conditions. For instance, in a task that involves sorting a list, the basic test cases verify that the list is sorted correctly for typical inputs. The second aspect ensures that the code performs well under edge scenarios, which are critical for evaluating the code's behavior under extreme or unusual conditions. These tests are designed to challenge the code with boundary conditions, unexpected inputs, and rare scenarios, to help in identifying potential bugs or weaknesses in the code that might not be evident during basic testing, such as using an empty list or a list with extremely large numbers to test the sorting algorithm. Finally, the test designer agent will also generate test cases with large scale values to assess the code's performance and scalability,, such as testing the sorting algorithm with a list of millions of elements. This involves testing the code under high-load conditions to evaluate whether it maintains its functionality and performance.

### 3.3 Test executor agent: code validation and feedback Integration

Distinct from the programmer agent and test designer agent that are powered by LLMs, the test executor agent in our framework is implemented through a Python script interacting with a local environment and the other two agents (an example of the test executor agent is shown in Appendix Figure 10). As illustrated in Fig. 1, the test executor agent plays a pivotal role in the final stage of the code generation process. Upon receiving code snippets generated by the programmer agent and test cases generated by the test designer agent, the test executor agent validates these code snippets along with the test cases in a local environment. The test executor agent closely monitors the return information from the execution environment (i.e., the terminal). This involves analyzing the output and determining whether the code snippets successfully pass all the test cases. If all test cases are passed, it returns the code to the human developer. Otherwise, if the execution results contain error information (e.g., syntax errors), the test executor agent will then return the error information to the programmer agent to fix the reported error.

## 4 Evaluation

In this section, we conduct experiments to answer the following research questions:

- RQ1: How does AgentCoder perform?
- RQ2: How do different agents contribute to the effectiveness of AgentCoder?
- RQ3: How do code refinement iterations affect AgentCoder's effectiveness?
- RQ4: How accurate are the tests generated by the test designer agent?
- RQ5: How adequate are the tests generated by the test designer agent?
- RQ6: Should the roles of programmer and test designer be separated and assigned to different agents?

### 4.1 Experiment Setup

We use pass@1 as the evaluation metric for code correctness, the most widely adopted metric in the literature of automatic code generation [Chen *et al.*, 2021c; Austin *et al.*, 2021; Dong *et al.*, 2023a; Zhang *et al.*, 2023b; Dong *et al.*, 2023b].

**Datasets.** In this paper, we evaluate AgentCoder's effectiveness with four widely used code generation datasets, i.e., HumanEval [Chen *et al.*, 2021b] and MBPP [Austin *et al.*, 2021], and their enhanced versions, i.e., HumanEval-ET and MBPP-ET [Dong *et al.*, 2023a]. HumanEval and HumanEval-ET focus on a range of programming challenges, offering a diverse set of problems to test the model's problem-solving skills and adaptability. On the other hand, MBPP and MBPP-ET provide a comprehensive collection of Python programming problems, designed to evaluate the model's proficiency in Python syntax and its ability to handle a variety

Table 1: End-to-end results of AgentCoder and baseline approaches for HumanEval, HumanEval-ET, MBPP, and MBPP-ET datasets. The best approach is highlighted in bold. The baseline results are obtained from its paper report. We use "-" to indicate the cases where the results are absent. The percentages in brackets are the improvement rate over the base LLMs (zero-shot prompting). For the last three rows, no baseline optimisation approaches report effectiveness on these LLMs, therefore, we report the results of AgentCoder only.

| Models | | | HumanEval | HumanEval-ET | MBPP | MBPP-ET | Mean |
|---|---|---|---|---|---|---|---|
| LLMs (zero-shot prompting) | | AlphaCode (1.1B) | 17.1 | - | - | - | 17.1 |
| | | Incoder (6.7B) | 15.2 | 11.6 | 17.6 | 14.3 | 14.7 |
| | | CodeGeeX (13B) | 18.9 | 15.2 | 26.9 | 20.4 | 20.4 |
| | | StarCoder (15.5B) | 34.1 | 25.6 | 43.6 | 33.4 | 34.2 |
| | | CodeGen-Mono (16.1B) | 32.9 | 25.0 | 38.6 | 31.6 | 32.0 |
| | | CodeX (175B) | 47.0 | 31.7 | 58.1 | 38.8 | 43.9 |
| | | CodeX (175B)+CodeT | 65.8 | 51.7 | 67.7 | 45.1 | 57.6 |
| | | GPT-3.5-turbo | 57.3 | 42.7 | 52.2 | 36.8 | 47.3 |
| | | PaLM Coder | 43.9 | 36.6 | 32.3 | 27.2 | 35.0 |
| | | Claude-instant-1 | 31.1 | 28.1 | 26.9 | 19.9 | 26.5 |
| | | GPT-4-turbo | 57.9 | 48.8 | 63.4 | 47.5 | 54.4 |
| | | GPT-4 | 67.6 | 50.6 | 68.3 | 52.2 | 59.7 |
| LLM-based optimisation approaches | with GPT-3.5-turbo | Few-Shot | 67.7 (18.2%) | 54.9 (28.6%) | 65.8 (26.1%) | 48.3 (31.2%) | 59.2 (25.2%) |
| | | CoT | 44.6 (-22.2%) | 37.2 (-12.9%) | 46.1 (-11.7%) | 34.8 (-5.4%) | 40.7 (-14.0%) |
| | | ReAct | 56.9 (-0.7%) | 49.4 (15.7%) | 67.0 (28.4%) | 45.9 (24.7%) | 54.8 (15.9%) |
| | | Reflexion | 68.1 (18.8%) | 50.6 (18.5%) | 70.0 (34.1%) | 47.5 (29.1%) | 59.1 (24.9%) |
| | | ToT | 54.4 (-5.1%) | 42.7 (0.0%) | 65.8 (26.1%) | 40.8 (10.9%) | 50.9 (7.6%) |
| | | RAP | 63.1 (10.1%) | 52.4 (22.7%) | 71.4 (36.8%) | 46.7 (26.9%) | 58.4 (23.5%) |
| | | Self-Edit | 62.2 (8.6%) | 54.3 (27.2%) | 56.4 (8.0%) | 45.9 (24.7%) | 54.7 (15.6%) |
| | | Self-Planing | 65.2 (13.8%) | 48.8 (14.3%) | 58.6 (12.3%) | 41.5 (12.8%) | 53.5 (13.1%) |
| | | Self-debugging | 61.6 (7.5%) | 45.8 (7.3%) | 60.1 (15.1%) | 52.3 (42.1%) | 55.0 (16.3%) |
| | | Self-Collaboration | 74.4 (29.8%) | 56.1 (31.4%) | 68.2 (30.7%) | 49.5 (34.5%) | 62.1 (31.3%) |
| | | INTERVENOR | 75.6 (31.9%) | 54.8 (28.3%) | 69.8 (33.7%) | 47.1 (28.0%) | 61.8 (30.7%) |
| | | CodeCoT | 79.3 (38.4%) | 69.5 (62.8%) | 89.5 (71.5%) | 63.0 (71.2%) | 75.3 (59.2%) |
| | | **AgentCoder (ours)** | **79.9 (39.4%)** | **77.4 (81.3%)** | **89.9 (72.2%)** | **89.1 (142.1%)** | **84.1 (77.8%)** |
| | with GPT-4 | Reflexion | 91.0 (34.6%) | - | 77.1 (12.9%) | - | 84.1 (40.9%) |
| | | Self-Debugging | - | - | 80.6 (18.0%) | - | 80.6 (35.0%) |
| | | Self-Collaboration | 90.2 (33.4%) | 70.7 (39.7%) | 78.9 (15.5%) | 62.1 (19.0%) | 75.5 (26.5%) |
| | | MetaGPT | 85.9 (27.1%) | - | 87.7 (28.4%) | - | 86.8 (45.4%) |
| | | **AgentCoder (ours)** | **96.3 (42.5%)** | **86.0 (70.0%)** | **91.8 (34.4%)** | **91.8 (75.9%)** | **91.5 (53.3%)** |
| | with PaLM Coder | **AgentCoder (ours)** | **64.0 (45.8%)** | **55.5 (51.6%)** | **75.9 (135.0%)** | **75.5 (177.6%)** | **67.7 (93.4%)** |
| | with Claude-instant-1 | **AgentCoder (ours)** | **67.7 (117.7%)** | **57.9 (106.0%)** | **76.3 (183.6%)** | **75.1 (277.4%)** | **69.3 (161.5%)** |
| | with GPT-4-turbo | **AgentCoder (ours)** | **89.6 (54.7%)** | **76.2 (56.1%)** | **91.4 (44.2%)** | **91.4 (92.4%)** | **87.2 (60.3%)** |

of coding scenarios. The enhanced versions, HumanEval-ET and MBPP-ET, include more adequate test cases, making them more challenging and better suited for evaluating advanced models.

**LLMs that power the agents.** The programmer agent and the test designer agent in AgentCoder are powered by LLMs. We study the effectiveness of AgentCoder powered by five state-of-the-art LLMs, including GPT-4, GPT-4-turbo, GPT-3.5-turbo, PaLM Coder, and Claude (Claude-instant-1).

**Baselines.** To illustrate the effectiveness of AgentCoder, we compare AgentCoder with 12 Large Language Models (LLMs), including open-source and closed-source ones, such as AlphaCode [Li *et al.*, 2022], Incoder [Fried *et al.*, 2022], CodeGeeX [Zheng *et al.*, 2023], StarCoder [Li *et al.*, 2023b], CodeGen-Mono [Nijkamp *et al.*, 2023b], CodeX [Brown *et al.*, 2020a], GPT-3.5-turbo, and GPT4 [OpenAI, 2023]. These models vary in architecture, training methodologies, and application scopes.

Additionally, we compare AgentCoder with 13 state-of-the-art (SOTA) code generation methods that are based on LLMs but with various optimisation strategies, including Few-shot learning, Chain-of-Thought [Wei *et al.*, 2022], ReAct [Yao *et al.*, 2022], Reflexion [Shinn *et al.*, 2023], ToT [Yao *et al.*, 2023], RAP [Hao *et al.*, 2023], Self-Edit [Zhang *et al.*, 2023b], Self-Planing [Jiang *et al.*, 2023], Self-Debugging [Chen *et al.*, 2023b], Self-Collaboration [Dong *et al.*, 2023b], SCOT [Li *et al.*, 2023a], CodeCoT [Huang *et al.*, 2023], and INTERVENOR [Wang *et al.*, 2023a]. These methods have been shown to significantly enhance the performance of LLMs in complex problem-solving scenarios.

Table 2: Contribution of different agents in AgentCoder.

| Agents | HumanEval | HumanEval-ET | MBPP | MBPP-ET |
|---|---|---|---|---|
| programmer agent only | 61.0 | 52.4 | 47.9 | 35.0 |
| programmer + test designer | 64.0 (11.7%) | 54.3 (27.2%) | 62.3 (19.3%) | 45.9 (24.7%) |
| programmer + test executor | 64.6 (12.7%) | 55.5 (30.0%) | 69.3 (32.8%) | 51.4 (39.7%) |
| AgentCoder | **79.9 (39.4%)** | **77.4 (81.3%)** | **89.9 (72.2%)** | **89.1 (142.1%)** |

## 4.2 RQ1: How does AgentCoder perform?

**Setup.** AgentCoder is a multi-agent-based code generation framework that can be applied to any LLM. To answer the first research question, we evaluate the effectiveness of AgentCoder with five state-of-the-art LLMs, i.e., GPT-4, GPT-4-turbo, GPT-3.5-turbo, PaLM Coder, and Claude (Claude-instant-1). As introduced in Section 4.1, we compare the pass@1 of AgentCoder with 12 LLMs and 13 LLM-based optimisation approaches that enhance the code generation performance with different strategies.

Tab. 1 shows the results. we can observe that AgentCoder outpeforms all the base LLM models and all the baseline optimisation approaches in all the datasets. Specifically, if we focus on the improvement that AgentCoder achieves over the base LLMs, take GPT-3.5-turbo as an example, GPT-3.5-turbo obtains 57.3% pass@1 in the HumanEval dataset, while AgentCoder obtains 79.9%. For GPT-4, the mean pass@1 of AgentCoder is 91.5% across all the datasets, **32.7% improvement** over the baseline zero-shot GPT-4 model. For PaLM Coder, Claude-instant-1, and GPT-4-turbo, the mean improvement of AgentCoder over the base models are **32.7%, 42.8%, 32.8%**, respectively.

AgentCoder also demonstrates superiority over all optimization baselines. For example, for MBPP-ET with GPT-3.5-turbo, AgentCoder obtains 89.1% pass@1, while CodeCoT, the state-of-the-art approach, achieves only 63.0%. On average, the pass@1 of AgentCoder is 84.1%, **8.8% more than the state-of-the-art approach CodeCoT**. One reason for AgentCoder's superiority over CodeCoT is that CodeCoT generates tests and code at the same time with only one agent, while AgentCoder has the test designer agent which generates more powerful test cases. RQ4 and RQ5 introduce more analysis on their comparison in terms of the effectiveness of test cases.

The HumanEval-ET and MBPP-ET datasets contain more comprehensive tests and are more challenging for code generation approaches to get high pass@1. We can observe that the base LLMs and the baseline optimisation approaches perform significantly worse on these two enhanced versions. However, AgentCoder's performance on these enhanced datasets is comparative to the original datasets, which is another superiority of AgentCoder, largely because the test designer agent generates rigorous tests to ensure that the generated code is indeed reliable.

## 4.3 RQ2: How do different agents contribute to the effectiveness of AgentCoder?

As illustrated in Fig. 1, AgentCoder contains three agents, i.e., the programmer agent, the test designer agent, and the test executor agent, where the programmer agent focuses on generating code snippets based on the code generation requirements and feedback from other agents. The test designer agent focuses on generating test cases, which are used to evaluate the correctness of the code snippets produced by the programmer agent. The test executor agent interacts with the other two agents to collect the code snippets and test cases and executes them in a local environment to prepare feedback. This research question investigates how each agent contributes to AgentCoder's effectiveness with four agent combination scenarios, i.e., the programmer agent itself, the programmer + test designer agent, where we feed the function and test cases into the programmer agent and require it to analyze whether it needs to refine the code to pass all test cases, and the programmer + test executor agent, where we directly run the generated code with the tests provided in the prompt [1](we provide the programmer + test designer/executor agent prompts in Appendix Figure 11 and 12).

The evaluation results are shown in Tab. 2. We can observe that first, with the assistant of the test designer and the test executor agent, the pass@1 increases compared with the result of only the programmer agent. For example, with both the programmer and the test designer agent, the pass@1 increases from 61.0% to 64.0%. However, without the test executor agent, the programmer agent is not able to get reliable feedback from dynamic test case execution. Therefore, the performance is significantly below AgentCoder. For the programer + test executor agent, it obtains 64.6% and 69.3% pass@1 in HumanEval and MBPP, which is also higher than the programmer agent itself which obtains 61.0% and 47.9%. This is because test executor agent detects some bugs in the code with the test cases provided by the prompt. However, the number of test cases is very limited, with only two to three tests in HumanEval and MBPP. The effectiveness of these tests are far below from the tests generated by the test designer agent. Therefore, without the test designer agent, the performance is also significantly below AgentCoder.

[1] The code generation prompts in HumanEval and MBPP contain a few test cases.

Table 3: Pass@1 of AgentCoder with different number of iterations on GPT-3.5-turbo.

| Iterations | HumanEval | HumanEval-ET | MBPP | MBPP-ET |
|---|---|---|---|---|
| 1 | 74.4 (29.8%) | 73.2 (71.4%) | 84.1 (61.1%) | 80.3 (118.2%) |
| 2 | 75.6 (31.9%) | 73.2 (71.4%) | 86.4 (65.5%) | 85.6 (132.6%) |
| 3 | 76.2 (33.0%) | 75.0 (75.6%) | 87.9 (68.4%) | 87.6 (138.0%) |
| 4 | 78.7 (37.3%) | 76.8 (79.9%) | 88.7 (69.9%) | 88.7 (141.0%) |
| 5 | 79.9 (39.4%) | 77.4 (81.3%) | 89.9 (72.2%) | 89.1 (142.1%) |

## 4.4 RQ3: How do code refinement iterations affect AgentCoder's effectiveness?

As illustrated in Fig. 1, AgentCoder will refine code snippets based on the feedback information provided by the test executor agent. In this experiment, we evaluate how the number of refinement iterations affect AgentCoder's effectiveness. Specifically, we analyze AgentCoder's effectiveness with its result for each refinement iteration. We can observe that the pass@1 increase with more iterations. In particular, when we increase the number of iterations from 1 to 5, the pass@1 of HumanEval and HumanEval-ET increases from 74.4% to 79.9% and 73.2% to 77.4%. We can also observe these behaviors for the MBPP and MBPP-ET datasets, where the pass@1 increases from 84.1% to 89.9% and 80.3% to 89.1%.

## 4.5 RQ4: How accurate are the tests generated by the test designer agent?

As we mentioned before, the test designer agent focuses on generating test cases to analyze whether the code has bugs and plays a crucial role in AgentCoder. However, once the test cases are incorrect (e.g., with incorrect test oracles), the feedback the test cases provide will be problematic, misleading the programmer agent and decreasing AgentCoder's overall effectiveness. Therefore, this research question investigates how reliable the test designer agent is in generating accurate tests to aid the programmer agent. We evaluate the accuracy of the test cases under the datasets' **canonical solution**[2] on GPT-3.5-turbo. The tests that pass the canonical solution are correct. To demonstrate the effectiveness and superiority of the test designer agent in AgentCoder, we compare the accuracy of the tests generated by AgentCoder, the default zero-shot GPT-3.5-turbo model, as well as CodeCoT where the tests are generated at the same time with the code.

[2] Each coding task in the datasets has a canonical solution, which is the ground truth for code generation.

Table 4: Accuracy of the generated test cases.

| Models | HumanEval | MBPP |
|---|---|---|
| GPT-3.5-turbo | 47.0 | 57.2 |
| CodeCoT | 67.1 | 79.0 |
| AgentCoder | 87.8 | 89.9 |

Table 5: Code line coverage of the generated test cases. In our experiment, we follow CodeCoT to calculate the code line coverage with the first five / all test cases provided by the test designer agent for each function.

| Models | HumanEval | MBPP |
|---|---|---|
| GPT-3.5-turbo | 67.1 / 70.2 | 58.4 / 61.3 |
| CodeCoT | 74.7 / 77.2 | 79.3 / 82.9 |
| AgentCoder | 84.7 / 87.5 | 85.3 / 89.5 |

The evaluation results are shown in Tab. 4. First, we observe that the accuracy of the tests cases produced by the test designer agent in AgentCoder is 87.8% and 89.9% respectively in HumanEval and MBPP datasets, while GPT-3.5-turbo obtains only 47.0% and 57.2%. In addition, we observe that the test designer agent in AgentCoder is also more accurate than CodeCoT in test generation. For example, on HumanEval, the accuracy is 87.8% v.s. 67.1% for AgentCoder and CodeCoT. The superiority of AgentCoder demonstrates the effectiveness of the prompt engineering strategies we designed for the test designer agent.

## 4.6 RQ5: How adequate are AgentCoder's test cases in terms of code coverage?

This research question explores the adequacy of the test cases generated by the test designer agent in terms of code coverage. Specifically, we evaluate how many lines of code in the canonical solution are covered by the test cases generated by the original GPT-3.5-turbo, CodeCoT, and AgentCoder. The evaluation results were illustrated in Tab. 5, where we can observe that the tests generated by AgentCoder have the highest code coverage. For example, AgentCoder obtains 84.7 / 87.5% and 85.3 / 89.5% code coverage compared with CodeCoT, which only obtains 74.7 / 77.2% and 79.3 / 82.9%, on the two datasets when we calculate the code line coverage with the first five / all tests generated by each strategy. The results further demonstrate the effectiveness of the prompt engineering strategies we adopt for the test designer agent.

## 4.7 RQ6: Should programmer and test designer be separated and assigned to different agents?

As shown in Fig. 1, AgentCoder requires separate agents for generating code and tests (i.e., the programmer agent and the test designer agent). Both agents are powered by LLMs. An alternative way is to let a single agent first generate code and then generate tests, within the same conversation. This research question investigates whether requiring one agent to finish two tasks, i.e., code generation and test case generation, is as effective as using separate agents.

The evaluation results are shown in Tab. 6, Tab. 7, and Tab. 8. We can observe that the pass@1 of using a single agent to generate both code and tests is lower than assigning the two tasks to different agents. For example, the pass@1 of the single agent has only 71.3% and 79.4% pass@1 for HumanEval and MBPP, while the multi-agent setup (AgentCoder) obtains 79.9% and 89.9% for HumanEval and MBPP. We also observe that the test case accuracy for the single agent is also lower than the multi-agent setting (AgentCoder). Specifically, the single agent only obtains 61.0% and 51.8% in HumanEval and MBPP datasets, while the multi-agent setup (AgentCoder) obtains 87.8% and 89.9% in HumanEval and MBPP. Finally, as shown in Tab. 8, we can also observe that the tests' coverage results of the single agent are also lower than in the multi-agent setup. For example, the single agent only obtains 72.5% and 75.9% code line coverage while multiple agents obtain 87.5% and 89.5% code line coverage.

Table 6: Pass@1 for a single agent and multiple agents.

| Models | HumanEval | HumanEval-ET | MBPP | MBPP-ET |
|---|---|---|---|---|
| Single Agent | 71.3 | 61.6 | 79.4 | 59.1 |
| Multiple Agents | **79.9** | **77.4** | **89.9** | **89.1** |

Table 7: Accuracy of the tests generated by single- and multi-agents.

| Models | HumanEval | MBPP |
|---|---|---|
| Single Agent | 61.0 | 51.8 |
| Multiple Agents | 87.8 | 89.9 |

Table 8: Code line coverage (with the first five / all test cases) of tests generated by single agent and multi-agent setup.

| Models | HumanEval | MBPP |
|---|---|---|
| Single Agent | 68.5 / 72.5 | 72.2 / 75.9 |
| Multiple Agents | 84.7 / 87.5 | 85.3 / 89.5 |

There are two possible reasons for the superiority of the multi-agent setup. First, letting a single agent do both code generation and test case design may distract the agent's focus; second, the tests designed by the same agent that generates the code can be biased by the code and lose objectivity, for example, if the generated code ignores the handling of edge cases, the generated tests can be affected by flaws in the code. These results demonstrate the necessity of using multiple agents to collaborate in code generation, with different agents taking different roles. Such benefit of multi-agent collaborations with LLMs has also been illustrated in other multi-agent systems [Chen *et al.*, 2023a; Zhang *et al.*, 2023a].

# 5 Conclusion

In this paper, we have proposed AgentCoder, which contains multiple agents to improve the code generation effectiveness of code generation models. AgentCoder contains three agents, i.e., the programmer, test designer, and test executor agent. During the code generation procedure, the programmer agent generates code snippets and then the test designer agent generates test cases for the code snippets. Next, the test executor agent tests the code snippets with test cases in the local environment. Once the feedback of the local environment contains an error message, the test executor agent feeds it into the programmer and test designer agent to require them fix the error information. Throughout our evaluations, AgentCoder demonstrated state-of-the-art performance, outperforming existing LLMs and prompt engineering methods in a variety of coding scenarios. For example, AgentCoder increases the pass@1 from 69.5% and 63.0% to 77.4% and 89.1% for HumanEval-ET and MBPP-ET datasets.

## References


[Ahmad *et al.*, 2021] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *ArXiv*, abs/2103.06333, 2021.

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

[Brown *et al.*, 2020a] 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, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *ArXiv*, abs/2005.14165, 2020.

[Brown *et al.*, 2020b] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[Chen *et al.*, 2021a] Jun Chen, Han Guo, Kai Yi, Boyang Albert Li, and Mohamed Elhoseiny. Visualgpt: Data-efficient adaptation of pretrained language models for image captioning. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 18009–18019, 2021.

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

[Chen *et al.*, 2021c] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura 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, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021.

[Chen *et al.*, 2023a] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Cheng Qian, Chi-Min Chan, Yujia Qin, Ya-Ting Lu, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *ArXiv*, abs/2308.10848, 2023.

[Chen *et al.*, 2023b] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *ArXiv*, abs/2304.05128, 2023.

[Dong *et al.*, 2023a] Yihong Dong, Ji Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. *ArXiv*, abs/2301.09043, 2023.

[Dong *et al.*, 2023b] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ArXiv*, abs/2304.07590, 2023.

[Feng *et al.*, 2020] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics.

[Fried *et al.*, 2022] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida I. Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *ArXiv*, abs/2204.05999, 2022.

[Guo *et al.*, 2020] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, and M. Zhou. Graphcodebert: Pre-training code representations with data flow. *ArXiv*, abs/2009.08366, 2020.

[Hao *et al.*, 2023] Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model. *ArXiv*, abs/2305.14992, 2023.

[Huang *et al.*, 2023] Dong Huang, Qi Bu, and Heming Cui. Codecot and beyond: Learning to program and test like a developer. *ArXiv*, abs/2308.08784, 2023.

[Jiang *et al.*, 2023] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. Self-planning code generation with large language model. *ArXiv*, abs/2303.06689, 2023.

[Le *et al.*, 2023] Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq R. Joty. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *ArXiv*, abs/2310.08992, 2023.

[Li *et al.*, 2022] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom, Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de, Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey, Cherepanov, James Molloy, Daniel Jaymin Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de, Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378:1092 – 1097, 2022.

[Li *et al.*, 2023a] Jia Li, Ge Li, Yongming Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. 2023.

[Li *et al.*, 2023b] 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, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nourhan Fahmy, Urvashi Bhattacharyya, W. Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jana Ebert, Tri Dao, Mayank Mishra, Alexander Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean M. Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! *ArXiv*, abs/2305.06161, 2023.

[Madaan *et al.*, 2023] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. *ArXiv*, abs/2303.17651, 2023.

[Nijkamp *et al.*, 2023a] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. *ICLR*, 2023.

[Nijkamp *et al.*, 2023b] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR*, 2023.

[OpenAI, 2023] OpenAI. Gpt-4 technical report. *ArXiv*, abs/2303.08774, 2023.

[Shinn *et al.*, 2023] Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. 2023.

[Wang *et al.*, 2021] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, 2021.

[Wang *et al.*, 2022] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Huai hsin Chi, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *ArXiv*, abs/2203.11171, 2022.

[Wang *et al.*, 2023a] Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. Intervenor: Prompt the coding ability of large language models with the interactive chain of repairing. *ArXiv*, abs/2311.09868, 2023.

[Wang *et al.*, 2023b] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

[Wei *et al.*, 2022] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Huai hsin Chi, F. Xia, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *ArXiv*, abs/2201.11903, 2022.

[Xi, 2020] Lin Xi. *Pareto Multi-task Learning and Its Applications*. PhD thesis, City University of Hong Kong, 2020.

[Yao *et al.*, 2022] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *ArXiv*, abs/2210.03629, 2022.

[Yao *et al.*, 2023] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *ArXiv*, abs/2305.10601, 2023.

[Zan *et al.*, 2022] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. CERT: Continual pre-training on sketches for library-oriented code generation. In *The 2022 International Joint Conference on Artificial Intelligence*, 2022.

[Zhang *et al.*, 2023a] Ceyao Zhang, Kaijie Yang, Siyi Hu, Zihao Wang, Guanghe Li, Yi Eve Sun, Chen Zhang, Zhaowei Zhang, Anji Liu, Song-Chun Zhu, Xiaojun Chang, Junge Zhang, F. Yin, Yitao Liang, and Yaodong Yang. Proagent: Building proactive cooperative ai with large language models. *ArXiv*, abs/2308.11339, 2023.

[Zhang *et al.*, 2023b] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. Self-edit: Fault-aware code editor for code generation. *ArXiv*, abs/2305.04087, 2023.

[Zheng *et al.*, 2023] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shanshan Wang, Yufei Xue, Zi-Yuan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie

Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *ArXiv*, abs/2303.17568, 2023.

# A Appendix

## A.1 Response Setup

To ensure that the output of each agent follows our requirements for the execution of the test executor agent, we will require each agent's output follow the architecture of `‘‘‘py[Code]’’’` and `‘‘‘py[TestCases]’’’`, where the [Code] and [TestCases] will be in the `‘‘‘py’’’`. With this format, the test executor agent can directly obtain [Code] and [TestCases] by removing the other sentences before and after these code blocks, ensuring an accurate and focused analysis.

## A.2 Case Illustration for CodeCoT and AgentCoder

To provide a comprehensive illustration for CodeCoT and AgentCoder, we provide two code and tests generation examples for HumanEval and MBPP datasets from Fig. 2 to Fig. 5. We can observe that AgentCoder can generate more fine-grained tests for the generated code. For example, AgentCoder will consider the code execution results when the input list does not contain element (Fig. 3 and Fig. 5), which can improve code snippet reliability for edge behaviors.

## A.3 Case Illustration on HumanEval dataset using AgentCoder

We also provide each agent's prompt and response example (Fig. 6 to Fig. 10) to illustrate AgentCoder's workflow. Fig. 6 and Fig. 7 illustrate AgentCoder's programmer prompt and response example. Fig. 8 and Fig. 9 provide AgentCoder's test designer prompt and response example. Fig. 10 illustrates AgentCoder's test executor source code.

## A.4 Case Illustration of the programmer + test executor agent

We illustrate the pipeline of the programmer + the test executor agent in Fig. 11.

## A.5 Case Illustration of the programmer + test designer

We illustrate the pipeline of the programmer + the test designer agent in Fig. 12.

**HumanEval Task 1 Code Generation**

```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """
    Check if in given list of numbers, are any two numbers closer to each other than given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

**CodeCoT**

```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """
    Check if in given list of numbers, are any two numbers closer to each other than given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
    for i in range(len(numbers)):
        for j in range(i + 1, len(numbers)):
            if numbers[i] - numbers[j] <= threshold:
                return True
    return False
```

**AgentCoder**

```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """
    Check if in given list of numbers, are any two numbers closer to each other than given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
 for i in range(len(numbers)):
        for j in range(i + 1, len(numbers)):
            if abs(numbers[i] - numbers[j]) <= threshold:
                return True
    return False
```

Figure 2: A case illustration of CodeCoT and AgentCoder generated code for HumanEval task. CodeCoT ignores to use of *abs()* function to check further the absolute values are lower than the threshold, while AgentCoder employs it to handle the negative values.

## HumanEval Task 1 Tests Generation

```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """
    Check if in given list of numbers, are any two numbers closer to each other than given
threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

## CodeCoT

```python
# Testing the function with various cases
assert not has_close_elements([1.0, 2.0, 3.0], 0.5)
# No two numbers are closer than 0.5, should return False

assert has_close_elements([1.0, 1.4, 2.5, 3.6], 0.5)
# 1.0 and 1.4 are closer than 0.5, should return True

assert not has_close_elements([], 1.0)
# Empty list, should return False

assert has_close_elements([0.1, 0.15, 0.3], 0.06)
# 0.1 and 0.15 are closer than 0.06, should return True

assert not has_close_elements([10, 20, 30, 40], 5)
# No two numbers are closer than 5, should return False
```

## AgentCoder

```python
# Basic Cases
# Test 1: Standard list with some close elements
assert has_close_elements([1.0, 2.5, 3.5, 5.0], 1.0) == True

# Test 2: Standard list with no close elements
assert has_close_elements([1.0, 3.0, 5.0, 7.0], 1.5) == False

# Edge Cases
# Test 1: Empty list
assert has_close_elements([], 1.0) == False

# Test 2: List with all elements the same
assert has_close_elements([3.0, 3.0, 3.0], 0.0) == True

# Test 3: Very small threshold
assert has_close_elements([1.0, 1.01, 2.0], 0.005) == False

# Test 4: List with only two elements
assert has_close_elements([1.0, 2.0], 1.5) == True

# Large Cases
large_list = [i * 0.1 for i in range(100000)]  # Creates a list [0, 0.1, 0.2, ..., 9999.9]

# Test with a threshold where we know the outcome
# Since the list is in increments of 0.1, a threshold of 0.05 should return False
assert has_close_elements(large_list, 0.05) == False

# Test with a larger threshold where we expect a True result
# With a threshold of 0.15, adjacent elements (0.1 apart) will be within the threshold
assert has_close_elements(large_list, 0.15) == True
```

Figure 3: A case illustration of CodeCoT and AgentCoder generated tests for HumanEval task. CodeCoT only considers the left values to be lower than the right values, which is due to the tests generated with its code where it also ignores the use of the *abs()* function, while AgentCoder considers two scenarios (i.e., left value lower/larger than the right values).

## MBPP Example Task Code Generation

Write a python function to check whether the given array is monotonic or not.

## CodeCoT

```
def is_Monotonic(array):
    """
    Check whether the given array is monotonic or not.
    :param array: List of integers
    :return: True if the array is monotonic, False otherwise
    """
    increasing = decreasing = True

    for i in range(1, len(array)):
        if array[i] > array[i - 1]:
            decreasing = False
        elif array[i] < array[i - 1]:
            increasing = False

    return increasing or decreasing
```

## AgentCoder

```
def is_Monotonic(arr):
    """
    Check whether the given array is monotonic or not.
    Args:
    arr (list): Input list of integers
    Returns:
    bool: True if the array is monotonic, False otherwise
    """

    # Check if array is increasing or decreasing
    increasing = decreasing = True

    for i in range(len(arr)-1):
        if arr[i] > arr[i+1]:
            increasing = False
        if arr[i] < arr[i+1]:
            decreasing = False

    # Check if either increasing or decreasing flag is True
    return increasing or decreasing
```

Figure 4: A case illustration of CodeCoT and AgentCoder generated code for MBPP task. Both CodeCoT and AgentCoder's code are correct. However, CodeCoT ignores the edge cases (e.g., the list does not contain values).

## MBPP Example Task Tests Generation

Write a python function to check whether the given array is monotonic or not.

## CodeCoT

```
# Self-examination with test cases
assert isMonotonic([1, 2, 3, 4, 5]) == True  # Increasing sequence, should return True
assert isMonotonic([5, 4, 3, 2, 1]) == True  # Decreasing sequence, should return True
assert isMonotonic([1, 2, 2, 3, 2]) == False  # Non-monotonic sequence, should return False
```

## AgentCoder

```
# Basic test cases
assert isMonotonic([1, 2, 3, 4, 5]) == True  # Increasing sequence
assert isMonotonic([5, 4, 3, 2, 1]) == True  # Decreasing sequence
assert isMonotonic([1, 2, 2, 3, 2]) == False  # Non-monotonic sequence

# Edge test cases
assert isMonotonic([1]) == True  # Single-element sequence
assert isMonotonic([]) == True  # Empty sequence
assert isMonotonic([1, 2]) == True  # Two-element increasing sequence
assert isMonotonic([2, 1]) == True  # Two-element decreasing sequence
assert isMonotonic([1, 1, 1, 1, 1]) == True  # All elements are equal, considered monotonic
assert isMonotonic([1, 1, 1, 2, 1]) == False  # Mixed values, not monotonic

# Large input
large_input = [i for i in range(1, 100001)]
# An increasing sequence of 100,000 elements
assert isMonotonic(large_input) == True  # Should be considered monotonic

# Create a large decreasing sequence
large_decreasing_input = [i for i in range(100000, 0, -1)]
# A decreasing sequence of 100,000 elements
assert isMonotonic(large_decreasing_input) == True  # Should be considered monotonic
```

Figure 5: A case illustration of CodeCoT and AgentCoder generated tests for MBPP task. CodeCoT ignores to consider the list does not contain values and in its generated code this scenario is also ignored. However, AgentCoder's edge cases will cover these edge scenarios.

## AgentCoder: Programmer Prompt

~~~~
**Role**: You are a software programmer.

**Task**: As a programmer, you are required to complete the function. Use a Chain-of-Thought approach to break down the problem, create pseudocode, and then write the code in Python language. Ensure that your code is efficient, readable, and well-commented.

For example:

**Input Code Snippet**:
```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """
    Check if in given list of numbers, are any two numbers closer to each other than given threshold.

    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
    # TODO: Implement the logic to determine if any two numbers are closer than the threshold
    pass

# Add your code here to complete the function
```

**Instructions**:
1. **Understand and Clarify**: Make sure you understand the task.
2. **Algorithm/Method Selection**: Decide on the most efficient way.
3. **Pseudocode Creation**: Write down the steps you will follow in pseudocode.
4. **Code Generation**: Translate your pseudocode into executable Python code.
~~~~

Figure 6: AgentCoder programmer prompt example.

## AgentCoder: Programmer Response

````
As a software developer, I'll follow the Chain-of-Thought approach outlined in the prompt to complete the
`has_close_elements` function. Let's break it down into the four steps:

1. **Understand and Clarify**:
   - The task is to write a function that checks if any two numbers in a given list are closer to each other than a
specified threshold.
   - The function should return `True` if such a pair exists, otherwise `False`.

2. **Algorithm/Method Selection**:
   - A straightforward way to solve this would be to compare each number with every other number in the list.
   - We can use a nested loop where each number is compared to every number that comes after it in the list.
   - The absolute difference between each pair will be calculated and compared to the threshold.

3. **Pseudocode Creation**:
   ```
   function has_close_elements(numbers, threshold):
      loop through each number in the list (i)
         for each number, loop through the remaining numbers in the list (j)
            if the absolute difference between numbers[i] and numbers[j] is less than or equal to the threshold
               return True
      if no such pair is found, return False
   ```

4. **Code Generation**:
   - Now, I'll translate the pseudocode into Python code.

Here's the complete function:

```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
   """
   Check if in given list of numbers, are any two numbers closer to each other than given threshold.

   >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
   False
   >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
   True
   """
   # Loop through each number in the list
   for i in range(len(numbers)):
      # For each number, loop through the remaining numbers
      for j in range(i + 1, len(numbers)):
         # If the absolute difference is less than or equal to the threshold, return True
         if abs(numbers[i] - numbers[j]) <= threshold:
            return True

   # If no such pair is found, return False
   return False
```
````

Figure 7: AgentCoder programmer response example.

## AgentCoder: Test Designer Prompt

**Role**: As a tester, your task is to create comprehensive test cases for the incomplete `has_close_elements` function. These test cases should encompass Basic, Edge, and Large Scale scenarios to ensure the code's robustness, reliability, and scalability.

**Input Code Snippet**:
```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """
    Check if in given list of numbers, are any two numbers closer to each other than given threshold.

    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```
**1. Basic Test Cases**:
- **Objective**: To verify the fundamental functionality of the `has_close_elements` function under normal conditions.

**2. Edge Test Cases**:
- **Objective**: To evaluate the function's behavior under extreme or unusual conditions.

**3. Large Scale Test Cases**:
- **Objective**: To assess the function’s performance and scalability with large data samples.

**Instructions**:
- Implement a comprehensive set of test cases following the guidelines above.
- Ensure each test case is well-documented with comments explaining the scenario it covers.
- Pay special attention to edge cases as they often reveal hidden bugs.
- For large-scale tests, focus on the function's efficiency and performance under heavy loads.

Figure 8: AgentCoder tester prompt example.

## AgentCoder: Test Designer Response

````
As a tester responding to the prompt, I will create and execute a series of Basic, Edge, and Large Scale Test Cases
for the `has_close_elements` function to ensure its robustness and reliability. Let's begin:

**1. Basic Test Cases:**

```python
# Test 1: Standard list with some close elements
assert has_close_elements([1.0, 2.5, 3.5, 5.0], 1.0) == True

# Test 2: Standard list with no close elements
assert has_close_elements([1.0, 3.0, 5.0, 7.0], 1.5) == False
```

**2. Edge Test Cases:**

```python
# Test 1: Empty list
assert has_close_elements([], 1.0) == False

# Test 2: List with all elements the same
assert has_close_elements([3.0, 3.0, 3.0], 0.0) == True

# Test 3: Very small threshold
assert has_close_elements([1.0, 1.01, 2.0], 0.005) == False

# Test 4: List with only two elements
assert has_close_elements([1.0, 2.0], 1.5) == True
```

**3. Large Scale Test Cases:**

For large-scale testing, I'll focus on the function's performance with a significantly large list. Due to the constraints
of this platform, I'll conceptualize the test case:

```python
# Large Scale Test 1: List with 100,000 elements in a predictable pattern
large_list = [i * 0.1 for i in range(100000)]  # Creates a list [0, 0.1, 0.2, ..., 9999.9]

# Test with a threshold where we know the outcome
# Since the list is in increments of 0.1, a threshold of 0.05 should return False
assert has_close_elements(large_list, 0.05) == False

# Test with a larger threshold where we expect a True result
# With a threshold of 0.15, adjacent elements (0.1 apart) will be within the threshold
assert has_close_elements(large_list, 0.15) == True
```
````

Figure 9: AgentCoder test designer response example.

## AgentCoder: Test Executor

```
from codegeex.benchmark.execution import check_correctness
def process_humaneval_test(sample, problems, example_test=False,language=language, test_case=True):
    task_id = sample["task_id"]
    task_id = problems.index(sample)
    prompt = sample["prompt"]
    code = sample["completion"]
    # Pre-process for different languages
    if language == "python" or language == "py":
        test_case = sample["test_list"]
        tests = ""
        for test in test_case:
            tests+="\n"+test
        test_string = code + "\n" + tests
    return test_string

def preprocess_data(task):
    if f"```py" in task["completion"]:
        task["completion"] = task["completion"][task["completion"].find(f"```py") +len(f"```py"):]
        task["completion"] = task["completion"][:task["completion"].find("```")]
    elif "```" in task["completion"]:
        task["completion"] = task["completion"][task["completion"].find("```") +3:]
        task["completion"] = task["completion"][:task["completion"].find("```")]
    return task

def test_report(dataset,lg):
    correct = 0
    for i in tqdm(range(len(dataset))):

        dataset[i]["full_code"] = process_humaneval_test(dataset[i], dataset,
example_test=False,language=lg,test_case=False)
        result = check_correctness(dataset[i]["task_id"],dataset[i],lg,5,"./tmp")
        if result["passed"]==True:
            correct+=1
        dataset[i]["report_passed"] = result["passed"]
        dataset[i]["report_result"] = result["result"]
    print("==============Start Report Testing==============")
    correct_percent = correct/len(dataset)*100
    print(f"test_report, {correct_percent:0.2f}")
    return dataset

def test_agent(dataset,lg):
    correct = 0
    for i in tqdm(range(len(dataset))):
        dataset[i]["full_code"] = process_humaneval_test(dataset[i], dataset,
example_test=False,language=lg,test_case=False)
        result = check_correctness(dataset[i]["task_id"],dataset[i],lg,5,"./tmp")
        if result["passed"]==True:
            correct+=1
        dataset[i]["result"] = result["result"]
        dataset[i]["passed"] = result["passed"]
    print("============Start Agent Testing=================")
    print("test_agent",correct)
    return dataset

model_list = ["gpt-3.5-turbo","palm-2-codechat-bison","claude-instant-1","gpt-4-1106-preview","gpt-4"]
language = ["py"]
for model_name in model_list:
    print(f"=================={model_name}================")
    epoch = 5
    path = AgentCoderProgrammerSaveResultPath
    with open(path, "r") as f:
        dataset = json.load(f)
    for current_epoch in range(epoch):
        with open(f"./dataset/{model_name}_{current_epoch}.json", "w") as f:
            json.dump(dataset, f)
        test_report(dataset,lg)
        test_agent(dataset,lg)
        dataset = call_completion(dataset,model_name,lg)
```

Figure 10: AgentCoder test executor script.

## Programmer+Test Designer

**Role**: As a programmer, you are required to complete the function `has_close_elements`. This function should check if any two numbers in a given list are closer to each other than a specified threshold.
Use a Chain-of-Thought approach to break down the problem, create pseudocode, and then write the final code in Python. Ensure that your code is efficient, readable, and well-commented.

**Input Code Snippet**:
```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """
    Check if in given list of numbers, are any two numbers closer to each other than given threshold.

    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
    for i in range(len(numbers)):
        for j in range(i + 1, len(numbers)):
            if abs(numbers[i] - numbers[j]) <= threshold:
                return True
    return False

# Test 1: Standard list with some close elements
assert has_close_elements([1.0, 2.5, 3.5, 5.0], 1.0) == True

# Test 2: Standard list with no close elements
assert has_close_elements([1.0, 3.0, 5.0, 7.0], 1.5) == False

# Test 1: Empty list
assert has_close_elements([], 1.0) == False

# Test 2: List with all elements the same
assert has_close_elements([3.0, 3.0, 3.0], 0.0) == True

# Test 3: Very small threshold
assert has_close_elements([1.0, 1.01, 2.0], 0.005) == False

# Test 4: List with only two elements
assert has_close_elements([1.0, 2.0], 1.5) == True

# Large Scale Test 1: List with 100,000 elements in a predictable pattern
large_list = [i * 0.1 for i in range(100000)]  # Creates a list [0, 0.1, 0.2, ..., 9999.9]

# Test with a threshold where we know the outcome
# Since the list is in increments of 0.1, a threshold of 0.05 should return False
assert has_close_elements(large_list, 0.05) == False

# Test with a larger threshold where we expect a True result
# With a threshold of 0.15, adjacent elements (0.1 apart) will be within the threshold
assert has_close_elements(large_list, 0.15) == True
```

**Instructions**:
1. **Understand and Clarify**: Make sure you understand the task. If necessary, write down what the function should do.
2. **Algorithm/Method Selection**: Decide on the most efficient way to compare the numbers in the list to find if any two are within the threshold.
3. **Pseudocode Creation**: Write down the steps you will follow in pseudocode. This should outline how you will iterate through the list and compare the numbers.
4. **Code Generation**: Translate your pseudocode into executable Python code. Remember to test your function with the provided examples and any additional cases you think are relevant.

Figure 11: Programmer + test executor example.

Programmer+Test Executor

```
import doctest

# Function definition as a string
text = """
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    '''
    Check if in given list of numbers, are any two numbers closer to each other than given threshold.

    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    '''
    for i in range(len(numbers)):
        for j in range(i + 1, len(numbers)):
            if abs(numbers[i] - numbers[j]) <= threshold:
                return True
    return False
"""

# Execute the function definition
exec(text)

# Run doctest
if __name__ == "__main__":
    doctest.testmod()
```

Figure 12: Programmer + test designer example.