CODESIM: Multi-Agent Code Generation and Problem Solving through Simulation-Driven Planning and Debugging

Md. Ashraful Islam[*1], **Mohammed Eunus Ali**[1], **Md Rizwan Parvez**[2]
[1]Bangladesh University of Engineering and Technology (BUET)
[2]Qatar Computing Research Institute (QCRI)
{mdashrafulpramanic, mohammed.eunus.ali}@gmail.com, mparvez@hbku.edu.qa

Abstract

Large Language Models (LLMs) have made significant strides in code generation and problem solving. Current approaches employ external tool-based iterative debuggers that use compiler or other tool-based runtime feedback to refine coarse programs generated by various methods. However, the effectiveness of these approaches heavily relies on the quality of the initial code generation, which remains an open challenge. In this paper, we introduce **CODESIM**, a novel multi-agent code generation framework that comprehensively addresses the stages of program synthesis—planning, coding, and debugging—through a human-like perception approach. As human verifies their understanding of any algorithms through visual simulation, CODESIM uniquely features a method of plan verification and internal debugging through the step-by-step simulation of input/output. Extensive experiments across seven challenging competitive problem-solving and program synthesis benchmarks demonstrate CODESIM's remarkable code generation capabilities. Our framework achieves new state-of-the-art (pass@1) results—(HumanEval **95.1%**, MBPP **90.7%**, APPS **22%**, and CodeContests **29.1%**). Furthermore, our method shows potential for even greater enhancement when cascaded with external debuggers. To facilitate further research and development in this area, we have open-sourced our framework in this link (https://kagnlp.github.io/codesim.github.io/).

1 Introduction

In recent years, the rise of Large Language Models (LLMs) has made significant advances in AI-assisted coding and reshaped the domain of code generation and problem-solving (Zhao et al., 2023). Code generation assistants built on GPT-4 (OpenAI, 2024), Mistral (Jiang et al., 2023a), and Llama (Dubey et al., 2024), inter alia, have demonstrated unprecedented ability to understand, generate, and manipulate code across various programming languages and problem domains. However, despite these advancements, significant challenges persist in generating code for complex programming tasks.

Current state-of-the-art approaches in code generation typically employ a *dual-pass* process (Shi et al., 2024; Jin et al., 2024b; Zhong et al., 2024; Levin et al., 2024). In the *first pass*, they use LLMs to generate an initial, fully/partially correct version of the program. Then accordingly in the *second pass*, they apply external tool-based iterative debuggers that leverage runtime compiler feedback or other diagnostic tools to refine and correct the generated code. While this approach has shown promise, it necessitates numerous iterations of LLM-tool interactions, and importantly its effectiveness is heavily dependent on the quality of the initial code generation—a process that continues to present substantial difficulties. Therefore, in this paper, we present **CODESIM**, a novel multi-agent code generation framework that seamlessly synthesizes complex code solutions without external resources, while offering potential for further enhancement through minimal external debugging.

Synthesizing programs even in the *first pass*, however, is fundamentally challenging, requiring a deep understanding of natural language processing, computer algorithms, data structures, and problem-solving strategies. These challenges are further compounded when attempting to generate code for competitive programming problems or advanced software engineering tasks, where adherence to specific constraints or passing unit tests are paramount (Khan et al., 2023).

While earlier code generation methods employed direct approaches (Chen et al., 2021a), chain-of-thought reasoning (Wei et al., 2022a), synthesized test-case guidance (Chen et al., 2022a), retrieval-augmented generation (Parvez et al.,

[*]Work done when working as a remote RA at QCRI.

Findings of the Association for Computational Linguistics: NAACL 2025, pages 5128–5154
April 29 - May 4, 2025

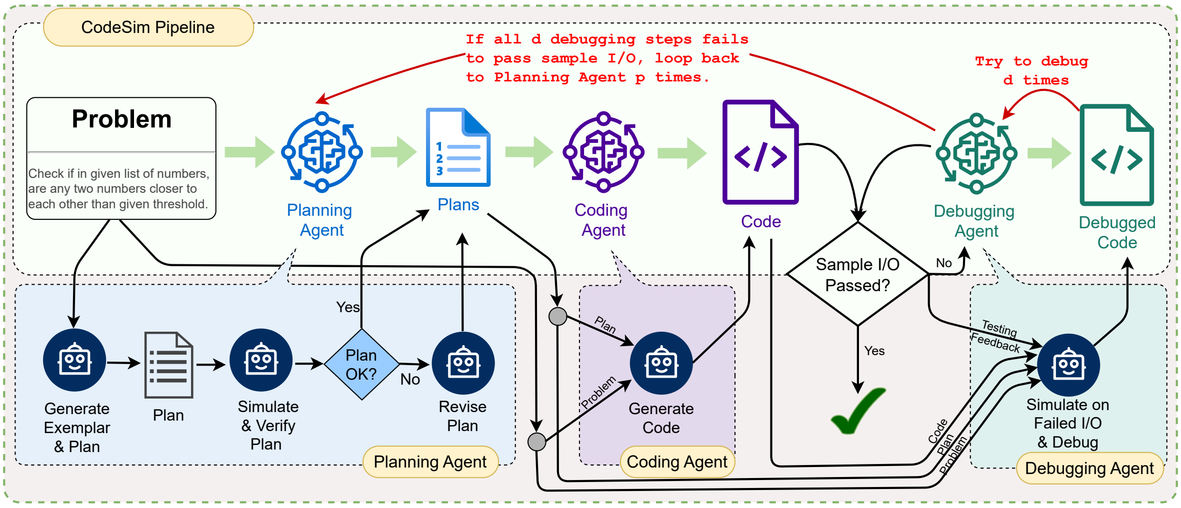

Figure 1: Overview of CODESIM: It consists of three agents—planning, coding, and debugging. The *Planning Agent* first generates an exemplar problem-solution (i.e., via self-retrieval) and devises a plan, which is then verified and refined through simulation. Next, the *Coding Agent* implements the plan. Finally, the *Debugging Agent* addresses potential bugs through step-wise simulation across d trials. The entire process iterates p times.

2021), and various in-context exemplar-based strategies (Shum et al., 2023; Zhang et al., 2022), recent paradigms have shifted toward plan-based (Jiang et al., 2023b), sampling or tree-searching (Zhou et al., 2023), self-retrieval (Yasunaga et al., 2023), and diverse agent-based approaches (Zhang et al., 2024; Qian et al., 2024; Shinn et al., 2023; Huang et al., 2023; Dong et al., 2023b).

Most recently, MapCoder (Islam et al., 2024a) proposes a multi-agent framework that implements agents emulating different stages of program synthesis such as recalling relevant examples, designing/planning, code generation, and testing/debugging. While this approach mimics a real developer's code generation cycle and shows improvements, it focuses solely on expanding steps without verifying the underlying hypotheses, with tests being performed only during the debugging phase. Consequently, the resulting gains are limited and it also requires larger number of iterations (i.e., LLM API calls).

To address these limitations, CODESIM—built upon planning, coding, and debugging agents—introduces a novel verification approach inspired by human problem-solving. By simulating input/output step-by-step, CODESIM verifies both the generated plans and performs internal debugging, mirroring how humans understand, visualize, and refine algorithms. This simulation-driven planning and debugging process ensures that each step is thoroughly evaluated, significantly enhancing both solution quality and efficiency. Figure 1 shows an overview of our proposed approach, CODESIM and in Figure 2, we demonstrate how simulation assists in both plan verification and debugging, highlighting its crucial role in improving problem-solving accuracy.

We evaluate CODESIM on seven popular programming synthesis benchmarks, including foundational tasks like HumanEval and MBPP, as well as challenging competitive problem-solving benchmarks such as APPS, and CodeContest. Our experiments leverage multiple LLMs, including ChatGPT, GPT-4, GPT-4o, LLaMa, Gemma, and Mixtral, showcasing significant improvements in their program synthesis capabilities. CODESIM consistently achieves state-of-the-art performances, often surpassing strong baselines like MapCoder. Additionally, our findings suggest that CODESIM's performance can be further improved when integrated with external debugging tools, such as LDB (Zhong et al., 2024), highlighting a promising direction for future research in hybrid code generation and debugging systems. Through detailed ablation studies, we provide valuable insights into CODESIM's functionality. We will open-source our framework to support future research in AI-assisted programming and problem-solving.

2 Related Work

Code Generation: Program synthesis has been a fundamental challenge in AI for decades (Manna and Waldinger, 1971). Early attempts with smaller language models centered on code generation by fine-tuning neural networks (Wang et al., 2021; Ahmad et al., 2021; Feng et al., 2020; Parvez et al., 2018; Hellendoorn and Devanbu, 2017; Rabinovich et al., 2017; Yin and Neubig, 2017; Hindle et al.,

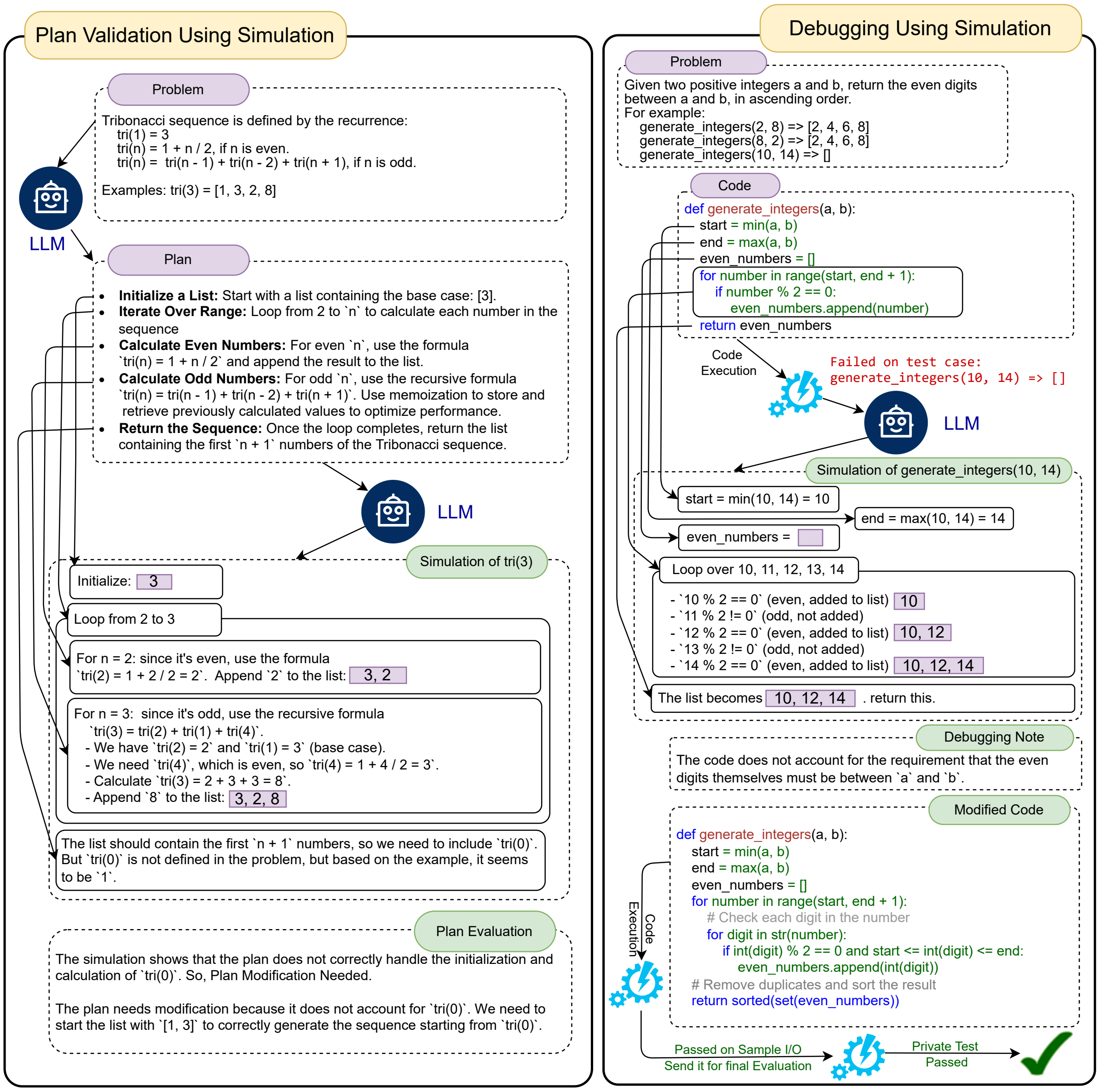

Figure 2: Example of Plan Validation using Simulation (left) and Debugging using Simulation (right) on two different problems using CODESIM.

2016), while others explored leveraging data flow information or conversational intents to guide the process (Andreas et al., 2020; Yu et al., 2019). Various prior approaches have also addressed code generation tasks using techniques such as data flow analysis and search-based methods (Li et al., 2022a; Parisotto and Salakhutdinov, 2017; Polozov and Gulwani, 2015; Gulwani, 2011).

LLMs for Code: Various LLMs have been developed for code synthesis (Austin et al., 2021; Chen et al., 2021b; Nijkamp et al., 2022; Fried et al., 2022; Allal et al., 2023; Li et al., 2022c). Recent open-source LLMs include the Llama family (Llama-2, CodeLlama, Llama3.1, etc.) (Roziere et al., 2023; Touvron et al., 2023), the Mistral family (Mistral, Mixtral, Codestral) (Jiang et al., 2023a), the Deepseek family (Deepseek Coder, Deepseek-V2, etc.) (Guo et al., 2024), MoTCoder (Li et al., 2023), and the Qwen family (Qwen 1.5, 2.5, 2.5-coder, etc.) (Hui et al., 2024), all of which are capable of solving many basic problems.

Prompting LLMs and Multi-Agent Code Generation: LLM prompting can be summarized into three categories: retrieval (Yasunaga et al., 2023; Parvez et al., 2021, 2023), planning (Jiang et al., 2023b; Wei et al., 2022b), and debugging (Le et al., 2022; Chen et al., 2022b, 2023; Ridnik et al., 2024), in addition to direct code generation approaches. In contrast, our work combines all these paradigms and bridges their gaps (See Table 1). Recently, nu-

Approach	Exemplars	Plan	Additional test cases generation	Debug	Simulation
Reflexion	✗	✗	✔	✔	✗
Self-planning	✗	✔	✗	✗	✗
Analogical	✔	✔	✗	✗	✗
LATS	✗	✔	✔	✔	✗
MapCoder	✔	✔	✗	✔	✗
CodeSim	✔	✔	✗	✔	✔

Table 1: Comparison of code generation approaches.

merous works have explored multi-agent code generation and problem-solving, including (Kulesza et al., 2004; Jin et al., 2024a; Phan et al., 2024), as well as approaches highlighted in Section 1. However, CODESIM uniquely features simulation-driven planning and LLM-based debugging. More recently, external debugging has emerged to further boost performance, such as LDB (Zhong et al., 2024), ChatDebug (Levin et al., 2024), and MGDebugger (Shi et al., 2024), which serve as a *second pass* after our generation.

3 CODESIM

Our goal is to develop a multi-agent code generation approach capable of complex problem solving. Drawing inspiration from recent works like MapCoder and ChatDev (in a different context), we devise the agents in CODESIM for planning, coding, and debugging. While these existing approaches focus primarily on expanding steps without verifying underlying hypotheses, we address this limitation by introducing a novel verification approach. Our approach simulates input/output step-by-step, verifying generated plans and performing internal debugging, mirroring how humans understand, visualize, and refine in algorithm development. Below, we present our proposed model.

3.1 Planning Agent

The first component of CODESIM is the *Planning Agent*. Given a problem description, the *Planning Agent* generates a single exemplar—a relevant problem along with its plan and solution. This mimics the behavior of human programmers, who, when faced with a new problem, first recall a similar problem they've previously solved. This exemplar-based recall is crucial as it provides a starting point for constructing a solution plan. Instead of generating multiple ungrounded exemplars as in MapCoder, our agent focuses on only one at a time. We then instruct the LLM to generate an appropriate plan. Once the plan is created, the LLM simulates (step-by-step) the solution with a sample input. If the simulation result does not match the expected output, the agent prompts the LLM to revise the plan. Otherwise, the plan is deemed valid. In the case of failure, the *Planning Agent* refines the plan. The complete prompts for the Planning Agent—including plan generation, verification, and refinement—are provided in the Appendix (Figure 5, 6, 7).

3.2 Coding Agent

Next component is the *Coding Agent*, which takes the problem description and the plan generated by the *Planning Agent* as input. The role of this agent is to translate the plan into executable code that solves the given problem. Once the code is generated, CODESIM evaluates it using sample input/output test cases. If the code passes all sample tests, it is returned as the final solution. Otherwise, the code is handed over to the next agent for further refinement. Figure 8 in the Appendix provides the complete prompt used by the *Coding Agent*.

3.3 Debugging Agent

The final component, the *Debugging Agent*, receives the original problem, the plan from the *Planning Agent*, the code generated by the *Coding Agent*, and the execution (unit testing) log as input to debug the code. To identify bugs, instead of directly prompting the LLMs, we uniquely leverage the simulation once again. The LLM is instructed specifically to simulate the code on inputs where it fails to produce the expected output, allowing it to trace the execution step by step and locate the error. Once the bug is identified, the LLM modifies the code to resolve the issue. The complete prompt for the *Debugging Agent* is shown in the Appendix (Figure 9). Unlike other approaches such as LATS (Zhou et al., 2023), AgentCoder (Huang et al., 2023), and Reflexion (Shinn et al., 2023), our *Debugging Agent* does not require additional test case generation. The rationale behind excluding this phase is discussed in the Ablation Study 6.8.

3.4 Adaptive Iteration

CODESIM employs an adaptive iteration starting with the *Planning Agent*, which generates plans for the given problem. These plans are passed to the *Coding Agent*, which translates them into code and tests against sample I/Os. If all tests pass, the code is returned; otherwise, it's sent to the *Debugging Agent*. The *Debugging Agent* attempts to fix the

Basic Programming Problems									
LLM	Approach	HumanEval	HumanEval ET	EvalPlus	Avg HumanEval	MBPP	MBPP-ET	Avg MBPP	Avg
ChatGPT	Direct	71.3%	64.6%	67.1%	67.7%	75.8%	52.6%	64.2%	65.9%
	CoT	70.7%	63.4%	68.3%	67.5%	78.3%	55.7%	67.0%	67.2%
	Self-Planning	70.7%	61.0%	62.8%	64.8%	73.8%	51.1%	62.5%	63.6%
	Analogical	67.1%	59.1%	59.1%	61.8%	69.3%	46.9%	58.1%	59.9%
	Self-collaboration	74.4%	56.1%	-	65.3%	68.2%	49.5%	58.9%	62.1%
	LATS	83.8%	-	-	-	-	-	-	-
	MapCoder	80.5%	70.1%	71.3%	74.0%	78.3%	54.4%	66.4%	70.2%
	CodeSim	86.0%	72.0%	73.2%	77.1%	86.4%	59.7%	73.1%	75.1%
GPT4	Direct	80.1%	73.8%	81.7%	78.5%	81.1%	54.7%	67.9%	73.2%
	CoT	89.0%	61.6%	-	75.3%	82.4%	56.2%	69.3%	72.3%
	Self-Planning	85.4%	62.2%	-	73.8%	75.8%	50.4%	63.1%	68.4%
	Analogical	66.5%	48.8%	62.2%	59.1%	58.4%	40.3%	49.4%	54.3%
	Reflexion	91.0%	78.7%	81.7%	83.8%	78.3%	51.9%	65.1%	74.4%
	LATS	92.7%	-	-	-	-	-	-	-
	MapCoder	93.9%	82.9%	83.5%	86.8%	83.1%	57.7%	70.4%	78.6%
	CodeSim	94.5%	81.7%	84.8%	87.0%	89.7%	**61.5%**	75.6%	81.3%
GPT4o	Direct	90.2%	81.1%	82.3%	84.5%	81.1%	55.9%	68.5%	76.5%
	CoT	90.9%	82.3%	**87.2%**	86.8%	82.9%	57.9%	70.4%	78.6%
	Self-Planning	89.0%	80.5%	84.1%	84.5%	82.60%	56.4%	69.50%	77.0%
	Analogical	88.4%	80.5%	83.5%	84.1%	75.10%	50.9%	63.00%	73.6%
	Reflexion	87.2%	81.1%	81.1%	83.1%	81.1%	56.7%	68.9%	76.0%
	LATS	88.8%	81.2%	-	85.0%	-	-	-	-
	MapCoder	90.2%	80.5%	81.7%	84.1%	88.7%	59.2%	74.0%	79.0%
	CodeSim	**95.1%**	**86.0%**	**87.2%**	**89.4%**	**90.7%**	61.2%	**76.0%**	**82.7%**

Table 2: Pass@1 results for different approaches on basic programming tasks.

code for up to d attempts. If unsuccessful after d attempts, the process returns to the *Planning Agent*, restarting the cycle. Once code passing all sample I/Os is obtained, the cycle ends, returning the code as the final output solution for evaluation against hidden test cases. This entire process repeats for a maximum of p cycles if needed. Algorithm 9 in the Appendix summarizes our adaptive agent traversal. The algorithm's complexity is $O(pd)$. Appendix 12 provides a comprehensive example of how CodeSim solves a problem.

4 Experimental Setup

4.1 Datasets

Following MapCoder, we evaluate CodeSim on five basic programming benchmarks i.e., **HumanEval** (Chen et al., 2021a), **HumanEval-ET** (Dong et al., 2023a), **EvalPlus** (Liu et al., 2023), **MBPP**) (Austin et al., 2021), and **MBPP-ET** (Dong et al., 2023a) and two competitive programming datasets i.e., **APPS** (Hendrycks et al., 2021), and **CodeContest** (Li et al., 2022b). For fair comparison, we collect all the datasets from the repository of the MapCoder.

4.2 Baselines and Metric

To evaluate CodeSim, we compare it against state-of-the-art code generation approaches, including **MapCoder**, as well as several notable methods: **Direct**, Chain of Thought (**CoT**) (Wei et al., 2022b), **Self-Planning** (Jiang et al., 2023b), **Analogical Reasoning** (Yasunaga et al., 2023), and **Self-collaboration** (Dong et al., 2023b). For simpler programming tasks, we include strong baselines such as Reflexion (Shinn et al., 2023) and LATS (Zhou et al., 2023). We exclude AgentCoder (Huang et al., 2023) due to reproducibility issues (discussed in Appendix 10). For fair comparison, our evaluation utilizes ChatGPT (gpt-3.5-turbo-1106), GPT-4 (gpt-4-1106-preview) from OpenAI, alongside open-source LLMs such as Gemma2-9B, Mixtral8x7B, LLaMa3.1-8B, and LLaMa3.1-70B. For basic programming tasks, we report next-generation performance with additional evaluations using GPT-4o (gpt-4o-2024-08-06). We adopt the widely used $pass@1$ metric, where a model is deemed successful if its sole predicted solution is correct.

4.3 Reproducibility

We aim to contribute to the NLP community by open-sourcing all of our code along with result logs, enabling others to reproduce our findings. For simple programming, we set the maximum number of planning tries to $p = 5$ and debugging tries to $d = 5$. For the competitive problem solving, we used $p = 3$ and $d = 3$ by default except for the CodeContest with GPT-4 where $p = 3$, $d = 5$.

5 Results

5.1 Basic Code Generation

In Table 2, we evaluate the model performances on simple code generation tasks. Overall, CODESIM demonstrates consistently superior performance compared to all other baselines across all datasets and LLMs. Notably, CODESIM achieves top scores with GPT-4o, reaching **95.1%** on HumanEval, **87.2%** on EvalPlus, and **90.7%** on MBPP, resulting in an impressive **82.7%** overall average and their new state-of-the-art (SoTA) results. This represents a significant improvement over the next best method, MapCoder, which scores 79.0% on average with GPT-4o. CODESIM's effectiveness is consistent across different model variants, outperforming other approaches with ChatGPT (**75.1% avg**) and GPT-4 (**81.3% avg**) as well. The method's robust performance across diverse datasets, including the challenging MBPP-ET where it achieves **61.5%** with GPT-4, underscores its versatility in handling various programming tasks. These results strongly indicate that CODESIM's simulation-driven planning and debugging approach marks a substantial advancement in code generation and problem-solving capabilities, as it consistently outperformed other baselines.

5.2 Competitive Problem Solving

In Table 3, we evaluate performance on complex, contest-level code generation tasks. CODESIM delivers significant improvements over other baselines in solving complex contest-level code generation tasks. With GPT-4, CODESIM reaches a strong **29.1%** on CodeContests and 22.0% on APPS, marking a consistent edge over MapCoder's 25.3% average. The performance gains are even more pronounced with ChatGPT, where CODESIM achieves a **16.4%** on CodeContests, and 12.0% on APPS resulting **14.2%** overall, outperforming MapCoder's 12.0%. These results highlight CODESIM's ability to handle the complexity of contest-level problems more effectively, especially through its simulation-driven approach.

LLM	Contest-Level Problems			
	Approach	CodeContest	APPS	Avg
ChatGPT	Direct	5.5%	8.0%	6.8%
	CoT	6.1%	7.3%	6.7%
	Self-Planning	6.1%	9.3%	7.7%
	Analogical	7.3%	6.7%	7.0%
	MapCoder	12.7%	11.3%	12.0%
	CodeSim	**16.4%**	**12.0%**	**14.2%**
GPT4	Direct	12.1%	12.7%	12.4%
	CoT	5.5%	11.3%	8.4%
	Self-Planning	10.9%	14.7%	12.8%
	Analogical	10.9%	12.0%	11.5%
	MapCoder	28.5%	**22.0%**	25.3%
	CodeSim	**29.1%**	**22.0%**	**25.6%**

Table 3: Pass@1 results for different approaches on CodeContest and APPS dataset.

Open-Source LLMs					
LLM	Approach	HumanEval	HumanEval ET	EvalPlus	Avg
Gemma2-9B	Direct	64.0%	56.1%	56.1%	58.7%
	CoT	31.7%	26.2%	27.4%	28.4%
	Reflexion	62.2%	56.7%	55.5%	58.1%
	CodeSim	82.9%	72.0%	72.6%	75.8%
Mixtral8x7B	Direct	20.7%	18.9%	18.9%	19.5%
	CoT	46.3%	42.1%	39.0%	42.5%
	Reflexion	34.1%	32.9%	29.9%	32.3%
	CodeSim	75.0%	61.6%	61.0%	65.9%
LLaMa3.1-8B	Direct	42.1%	38.4%	39.0%	39.8%
	CoT	48.8%	42.1%	43.3%	44.7%
	Reflexion	43.9%	31.1%	29.9%	35.0%
	CodeSim	79.9%	65.2%	61.2%	68.8%
LLaMa3.1-70B	Direct	57.3%	50.6%	52.4%	53.4%
	CoT	75.6%	67.7%	70.1%	71.1%
	Reflexion	73.8%	64.0%	68.3%	68.7%
	CodeSim	**90.2%**	**73.8%**	**76.2%**	**80.1%**

Table 4: Pass@1 results for different approaches using Open-source LLMs.

5.3 Performance Across Open-source LLMs

To further demonstrate CODESIM's generalization capability, we evaluate its performance with open-source LLMs, including Gemma2-9B, Mixtral8x7B, LLaMa3.1-8B, and LLaMa3.1-70B. As shown in Table 4, CODESIM consistently outperforms all other methods across these models. On LLaMa3.1-70B, CODESIM achieves an accuracy of **90.2%** on HumanEval and **76.2%** on EvalPlus, with an average of **80.1%**, closely matching GPT-4o's performance. Due to the complex prompting scheme of MapCoder, open-source LLMs often struggle to generate output in the correct format. Therefore, we exclude MapCoder from this experiment. On the other hand, Reflexion shows minimal improvement in accuracy. These results highlight CODESIM's strong generalization ability across various LLM architectures, even on smaller models like Gemma2-9B that achieves a notable avg accuracy of **75.8%**.

6 Ablation Studies and Analyses

6.1 Impact of Different Agents

Our primary contributions are two folds: (i) the simulation-guided plan verification step within the *Planning Agent* and (ii) the bug fixing process through simulation in *Debugging Agent*. To evaluate the significance of these components, we ablate these two parts of our approach and present the results in Table 5. The findings confirm that both components contribute significantly.

Simulation Driven Planning	Debugging using Simulation	Pass@1	Performance Drop
✗	✗	92.1%	3.2%
✔	✗	93.3%	1.9%
✗	✔	93.3%	1.9%
✔	✔	95.1%	-

Table 5: Pass@1 results for different versions of CODESIM (by using GPT4o on HumanEval dataset).

6.2 Fine-grained Analysis of the Impact of Simulation

Table 6 presents the impact of incorporating *Simulation* in CODESIM. The results show that CODESIM consistently outperforms other approaches across both simple and multi-agent settings, demonstrating superior performance with both open-source and proprietary LLMs. This highlights the effectiveness of *Simulation* in enhancing problem-solving efficiency within our pipeline.

6.3 Impact of Varying Programming Languages

To evaluate the performance of CODESIM across various programming languages, we utilized the xCodeEval (Khan et al., 2023) dataset. The experimental results, presented in Table 7, demonstrate that CODESIM maintains strong performance across different programming languages, highlighting its versatility and effectiveness.

LLM	Method	Approach	HumanEval (Pass@1)	Impact of using Simulation
GPT4o	Simpler	Direct	90.2%	5.4%
		CoT	90.9%	4.6%
		Self-Planning	89.0%	6.9%
		Analogical	88.4%	7.6%
		Reflexion	91.0%	4.5%
		LATS	88.8%	7.1%
	Multi-Agent	MapCoder	90.2%	5.4%
		CodeSim	95.1%	-
LLaMa3.1-70B	Simpler	Direct	57.3%	57.4%
		CoT	75.6%	19.3%
		Reflexion	73.8%	22.2%
	Multi-Agent	CodeSim	90.2%	-

Table 6: Impact of using *Simulation*.

Dataset	Language	Direct	MapCoder	CodeSim
xCodeEval	Python	17.9%	**27.4%**	**27.4%**
	C	9.4%	21.7%	**24.5%**
	Rust	12.3%	21.7%	**23.6%**

Table 7: Pass@1 results for different programming languages from xCodeEval dataset by using ChatGPT.

6.4 Use of External Debugger

LLM	LDB	Reflexion	MapCoder	CodeSim
ChatGPT	without	88.0%	90.2%	95.1%
	with	89.6%	92.1%	96.3%
GPT-4o	without	88.0%	90.2%	95.1%
	with	94.5%	91.5%	**97.6%**

Table 8: Pass@1 results for different approaches using an external debugger.

The performance of CODESIM can be further enhanced by incorporating an external debugger in the *second pass*. We experiment with LDB as the external debugger on HumanEval dataset in Table 8. We use the output code from the most competitive *first-pass* generation methods, including CODESIM, Reflexion, and MapCoder, using GPT-4o as the backbone. These seed programs are then passed to LDB, which was tested with two different LLMs: ChatGPT and GPT-4o. As can be seen, CODESIM achieves 95.1% accuracy in the *first pass* with GPT-4o, surpassing Reflexion's *second pass* performance of 94.5%. By utilizing LDB with GPT-4o, CODESIM achieves a *second pass* accuracy of 97.6%, setting a new state-of-the-art result for a *dual-pass* approach. In addition, we note that the *second pass* with LDB consumes 39K more tokens in Reflexion compared to our approach, highlighting the efficiency of CODESIM.

6.5 Qualitative Example

We also conduct a qualitative analysis to better understand how CODESIM improves performance across various datasets. Figure 2 demonstrates how CODESIM enhances the plan through simulation and assists in debugging the code using the same technique. A complete example, including LLM output, is provided in Appendix 12.

6.6 Impact of p and d

CODESIM includes two key hyperparameters: the maximum number of planning steps (p) and the maximum number of debugging steps (d). By varying these parameters, we plot the results in Figure 3, which shows a proportionate improvement in performance. It is important to note that higher values of p and d lead to more API calls and increased token consumption, allowing users to adjust these parameters to balance between accuracy and cost.

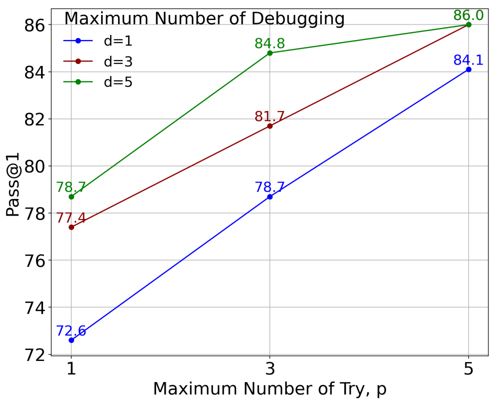

Figure 3: Pass@1 results by varying maximum number of planning, p and maximum number of debugging, d.

6.7 Impact of Number of Sample I/Os

The HumanEval dataset has an average of only 2.82 sample I/Os per example, which is a relatively small number for deriving meaningful insights. In this ablation, we augment the dataset by adding 5 more sample I/Os from the HumanEval-ET dataset. This augmentation increases performance notably, leading to 89% accuracy with ChatGPT, a 3.5% improvement over previous results, 86%.

6.8 Impact of Synthesizing Additional I/O

Increasing the number of sample I/Os for testing can enhance the overall performance of our approach, as indicated in 6.7. Based on this insight, we use a self-consistency (Wang et al., 2023a) method to generate additional test cases. We instruct the LLM to generate five more test cases for each problem, covering both basic and edge cases. The LLM is called twice, and we select the test cases that are present in both responses. However, this approach results in a performance decline. With ChatGPT we achieve 78% accuracy—a 9.3% decrease from the original 86%. This indicates that generating additional I/Os is a non-trivial task that may negatively impact final outcomes.

6.9 API Call and Token Analysis

We compare the API calls and token consumption of our approach with the previous state-of-the-art method, MapCoder (Islam et al., 2024a), as shown in Table 9. The results reveal that CODESIM not only improves performance but also reduces token consumption. On average, CODESIM uses 4.13 thousand fewer tokens while achieving a 7.1% increase in accuracy, proving that CODESIM is more efficient in both accuracy and token usage compared to MapCoder.

6.10 Error Analysis and Challenges

Although CODESIM demonstrates strong performance compared to other methods, it faces challenges in specific algorithmic domains. The APPS dataset (Hendrycks et al., 2021) includes problems with three levels of difficulty: (i) Introductory, (ii) Interview, and (iii) Competition. Figure 4 illustrates the performance of different approaches based on difficulty level. The results indicate that for introductory and interview-level problems, CODESIM does not surpass MapCoder when using ChatGPT. Additionally, when using GPT-4, CODESIM struggles to outperform MapCoder on interview-level problems. Upon manual review, we observe that for more complex issues, such as dynamic programming (DP), CODESIM encounters difficulties in constructing the DP table.

LLM	Dataset	Average API Calls ↓		Average Token Consumption (K) ↓		Token Reduction over MapCoder (k) ↑	Acc Gain over MapCoder ↑
		MapCoder	CodeSim	MapCoder	CodeSim		
ChatGPT	HumanEval	17	7	10.41	5.48	4.93	6.8%
	MBPP	12	6	4.84	4.24	0.60	10.3%
	APPS	21	15	26.57	19.20	7.37	6.2%
	CodeContest	23	16	34.95	24.02	10.92	29.1%
GPT4	HumanEval	15	5	12.75	5.15	7.60	0.6%
	MBPP	8	5	4.96	5.21	-0.26	7.9%
	APPS	19	13	31.80	23.18	8.61	0.0%
	CodeContest	19	17	38.70	41.66	-2.95	2.1%
GPT4o	HumanEval	9	4	6.63	3.84	2.79	5.4%
	MBPP	9	5	6.10	4.43	1.67	2.3%
Average		15.2	9.3	17.77	13.64	4.13	7.1%

Table 9: Comparison between MapCoder and CODESIM in terms of average number of API calls, average tokens used (in thousands). Here the upward symbol (↑) refers that the higher value is better and opposite meaning for downward symbol (↓).

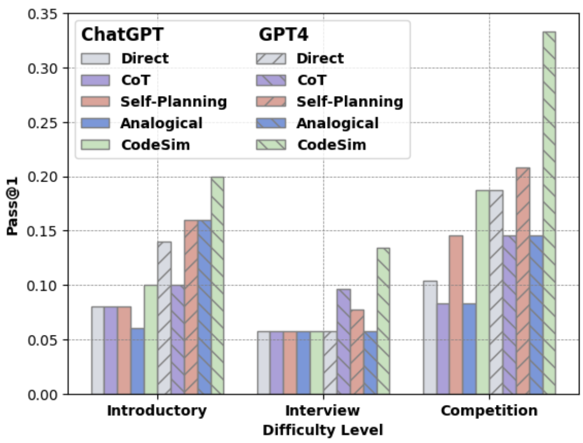

Figure 4: Performance of different approaches across different difficulty levels on the APPS dataset.

7 Conclusion and Future Work

In this paper, we introduce CODESIM, a novel framework that leverages the multi-agent prompting capabilities of LLMs for efficient code generation in problem-solving tasks. CODESIM integrates three agents—planning, coding, and debugging—to effectively solve programming problems. It harnesses the power of simulation for plan verification and debugging, significantly outperforming existing state-of-the-art approaches by a wide margin. Future work will focus on extending this approach to other domains such as mathematical reasoning and question answering broadening its scope and impact.

8 Limitations

In Section 6.4, we observe that utilizing an external debugger can further enhance our results. Our next research goal is to achieve the best performance without relying on any external tools. Although we have reduced token consumption compared to the previous state-of-the-art method, MapCoder, it still remains high compared to the direct prompting approach. Direct prompting consumes an average of 560 tokens, while our method consumes around 13,640 tokens. This indicates room for enhancement in efficiency. While in this work, we generate the exemplars with the LLMs themselves, in general they are found from external resource (Parvez and Chang, 2021). Although this has its own challenges such as noisy retrievals (Wang et al., 2023b), inconsistent generations (Islam et al., 2024b; Parvez, 2024; Sadat et al., 2023) this direction could also be a possible improvement. Another limitation is the use of external tools for assistance during simulation. We have not explored this avenue in the current research, leaving it for future work. Additionally, more sample I/Os could potentially improve performance, and our future research will focus on investigating methods for generating accurate additional I/Os. Moreover, we would like to note that in this work, we focus solely on generated code's correctness and did not study its optimizations such as test-time, memory. Finally, it is advisable to run the machine generated code inside a sandbox to avoid any potential risks.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz

Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, Hao Fang, Alan Guo, David Hall, Kristin Hayes, Kellie Hill, Diana Ho, Wendy Iwaszuk, Smriti Jha, Dan Klein, Jayant Krishnamurthy, Theo Lanman, Percy Liang, Christopher H. Lin, Ilya Lintsbakh, Andy McGovern, Aleksandr Nisnevich, Adam Pauls, Dmitrij Petters, Brent Read, Dan Roth, Subhro Roy, Jesse Rusak, Beth Short, Div Slomin, Ben Snyder, Stephon Striplin, Yu Su, Zachary Tellman, Sam Thomson, Andrei Vorobev, Izabela Witoszko, Jason Wolfe, Abby Wray, Yuchen Zhang, and Alexander Zotov. 2020. Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics*, 8:556–571.

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

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022a. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022b. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri 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, Josh 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. 2021a. Evaluating large language models trained on code.

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. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. 2023a. Codescore: Evaluating code generation by learning code execution. *arXiv preprint arXiv:2301.09043*.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023b. Self-collaboration code generation via chatgpt.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and et al. 2024. The llama 3 herd of models.

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

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. *arXiv preprint arXiv:2204.05999*.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 763–773, New York, NY, USA. ACM.

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

Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM*, 59(5):122–131.

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024a. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, Bangkok, Thailand. Association for Computational Linguistics.

Shayekh Bin Islam, Md Asib Rahman, K S M Tozammel Hossain, Enamul Hoque, Shafiq Joty, and Md Rizwan Parvez. 2024b. Open-RAG: Enhanced retrieval augmented reasoning with open-source large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 14231–14244, Miami, Florida, USA. Association for Computational Linguistics.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023a. Mistral 7b.

Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023b. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689*.

Dongming Jin, Zhi Jin, Xiaohong Chen, and Chunhui Wang. 2024a. Mare: Multi-agents collaboration framework for requirements engineering. *arXiv preprint arXiv:2405.03256*.

Haolin Jin, Zechao Sun, Yiheng Yang, and Huaming Chen. 2024b. Rgd: Multi-llm based agent debugger via refinement and generation guidance. *arXiv preprint arXiv:2410.01242*.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*.

Uirá Kulesza, Alessandro Garcia, Carlos Lucena, and Paulo Alencar. 2004. A generative approach for multi-agent system development. In *International Workshop on Software Engineering for Large-Scale Multi-agent Systems*, pages 52–69. Springer.

Md Tahmid Rahman Laskar, Sawsan Alqahtani, M Saiful Bari, Mizanur Rahman, Mohammad Abdullah Matin Khan, Haidar Khan, Israt Jahan, Amran Bhuiyan, Chee Wei Tan, Md Rizwan Parvez, Enamul Hoque, Shafiq Joty, and Jimmy Huang. 2024. A systematic survey and critical review on evaluating large language models: Challenges, limitations, and recommendations. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 13785–13816, Miami, Florida, USA. Association for Computational Linguistics.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Kyla Levin, Nicolas van Kempen, Emery D Berger, and Stephen N Freund. 2024. Chatdbg: An ai-powered debugging assistant. *arXiv preprint arXiv:2403.16354*.

Jingyao Li, Pengguang Chen, and Jiaya Jia. 2023. Motcoder: Elevating large language models with modular of thought for challenging programming tasks. *arXiv preprint arXiv:2312.15960*.

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

Yujia Li, David 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 J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022b. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Yujia Li, David 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 Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022c. Competition-level code generation with alphacode.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Zohar Manna and Richard J. Waldinger. 1971. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165.

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. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2024. Gpt-4 technical report.

Emilio Parisotto and Ruslan Salakhutdinov. 2017. Neural map: Structured memory for deep reinforcement learning. *arXiv preprint arXiv:1702.08360*.

Md Rizwan Parvez. 2024. Evidence to generate (e2g): A single-agent two-step prompting for context grounded and retrieval augmented reasoning. *arXiv preprint arXiv:2401.05787*.

Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601*.

Md Rizwan Parvez, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2018. Building language models for text with named entities. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2373–2383, Melbourne, Australia. Association for Computational Linguistics.

Md Rizwan Parvez and Kai-Wei Chang. 2021. Evaluating the values of sources in transfer learning. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5084–5116, Online. Association for Computational Linguistics.

Md Rizwan Parvez, Jianfeng Chi, Wasi Uddin Ahmad, Yuan Tian, and Kai-Wei Chang. 2023. Retrieval enhanced data augmentation for question answering on privacy policies. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 201–210, Dubrovnik, Croatia. Association for Computational Linguistics.

Huy Nhat Phan, Phong X Nguyen, and Nghi DQ Bui. 2024. Hyperagent: Generalist software engineering agents to solve coding tasks at scale. *arXiv preprint arXiv:2409.16299*.

Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, Bangkok, Thailand. Association for Computational Linguistics.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. *CoRR*, abs/1704.07535.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*.

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

Mobashir Sadat, Zhengyu Zhou, Lukas Lange, Jun Araki, Arsalan Gundroo, Bingqing Wang, Rakesh Menon, Md Parvez, and Zhe Feng. 2023. DelucionQA: Detecting hallucinations in domain-specific question answering. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 822–835, Singapore. Association for Computational Linguistics.

Yuling Shi, Songsong Wang, Chengcheng Wan, and Xiaodong Gu. 2024. From code to correctness: Closing the last mile of code generation with hierarchical debugging. *arXiv preprint arXiv:2410.01215*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Kashun Shum, Shizhe Diao, and Tong Zhang. 2023. Automatic prompt augmentation and selection with chain-of-thought from labeled data. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 12113–12139, Singapore. Association for Computational Linguistics.

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

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023a. Self-consistency improves chain of thought reasoning in language models.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Zhiruo Wang, Jun Araki, Zhengbao Jiang, Md Rizwan Parvez, and Graham Neubig. 2023b. Learning to filter context for retrieval-augmented generation. *arXiv preprint arXiv:2311.08377*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022a. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022b. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H Chi, and Denny Zhou. 2023. Large language models as analogical reasoners. *arXiv preprint arXiv:2310.01714*.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *CoRR*, abs/1704.01696.

Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter Lasecki, and Dragomir Radev. 2019. CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1962–1979, Hong Kong, China. Association for Computational Linguistics.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658, Bangkok, Thailand. Association for Computational Linguistics.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870, Bangkok, Thailand. Association for Computational Linguistics.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

Appendix

9 Algorithm of CodeSim

Algorithm 1 shows the pseudo-code of our prompting technique.

Algorithm 1 CodeSim

```
p ← maximum number of planning steps
d ← maximum number of debugging steps

for i ← 1 to p do
    # Start of Planning Agent
    plan ← GeneratePlan(problem)
    feedback ← ValidatePlan(problem, plan)
    if feedback is negative then
        plan ← RefinePlan(problem, plan, feedback)
    end if
    # End of Planning Agent

    # Start of Coding Agent
    code ← GenerateCode(problem, plan)
    passed, log ← test(code, sample_io)
    if passed then
        Return code
    else
        # Start of Debugging Agent
        for j ← 1 to d do
            code ← DebugCode(
              problem,
              plan,
              code,
              log
            )

            passed, log ← test(code, sample_io)
            if passed then
                Return code
            end if
        end for
        # End of Debugging Agent
    end if
    # End of Coding Agent

end for
Return code
```

10 Exclusion of AgentCoder

We have not included AgentCoder (Huang et al., 2023) in our comparison due to reproducibility issues which undoubtedly plays a critical role in fair comparison as indicted in Laskar et al. (2024), as we were unable to replicate their results. In our attempts to reproduce their work on the HumanEval benchmark using ChatGPT, we achieved 56.7% accuracy after four iterations, consuming 11.9 million tokens. When using GPT-4, we attained only 17.7% accuracy after two iterations, with 10.4 million tokens consumed. The token consumption in both cases is significantly higher compared to MapCoder (1.7 million tokens with ChatGPT and 2.1 million with GPT-4) and CodeSim(0.89 million tokens in ChatGPT and 0.85 million in GPT-4). These two experiments resulted in a cost of approximately \$500 USD, but we were still unable to come close to AgentCoder's reported claims of 79.9% accuracy with ChatGPT and 96.3% with GPT-4.

Furthermore, we found unaddressed issues on their GitHub page (link) related to reproducibility. Additionally, for the MBPP dataset, they used all test cases as public test cases (link), which deviates from standard practices. As a result, we did not consider those results in our comparison either.

11 Details Promptings of CodeSim

The *Planning Agent* interacts with the LLM three times to generate a plan. In the first API call, it instructs the LLM to comprehend the problem, generate an example problem, recommend a suitable algorithm, and finally produce the plan (Figure 5). In the second API call, the LLM is instructed to verify the plan through simulation (Figure 6). If the plan is satisfactory, it is returned by the agent. Otherwise, the LLM is called again to refine the plan based on the feedback from the simulation (Figure 7).

The next step involves the *Coding Agent*, which receives the plan from the *Planning Agent* and uses the prompt outlined in Figure 8 to generate code.

If the code fails to pass the sample input/output, CodeSim activates its final agent, the *Debugging Agent*, using the prompt shown in Figure 9.

These figures also include the rationale behind the inclusion of each sentence in the prompt.

12 Example Problem

We present a complete example of problem solving using CodeSim below:

Coding Agent: Prompt for Plan Generation

```
You are a programmer tasked with generating appropriate plan to solve a given problem
using the **{language}** programming language.

## Problem
{problem}

**Expected Output:**
Your response must be structured as follows:

### Problem Understanding
- Think about the original problem. Develop an initial
  understanding about the problem.

### Recall Example Problem
Recall a relevant and distinct problems (different from problem
mentioned above) and
- Describe it
- Generate {language} code step by step to solve that problem
- Discuss the algorithm to solve this problem
- Finally generate a planning to solve that problem

### Algorithm to solve the original problem
- Write down the algorithm that is well suited for the original
  problem
- Give some tutorials to about the algorithm for example:
    - How to approach this type of algorithm
    - Important things to consider

### Plan
- Write down a detailed, step-by-step plan to solve the
  **original problem**.

--------
**Important Instruction:**
- Strictly follow the instructions.
- Do not generate code.
```

Allow the LLM sufficient time and space to process and comprehend the problem.

Rather than providing the LLM with a predefined example, we leverage its inherent knowledge to independently recall a relevant problem that can aid in solving the original issue.

Guide the LLM to determine the type of algorithm suitable for solving the problem, and request a tutorial or explanation on how to apply it.

Lastly, instruct the LLM to generate a detailed plan to solve the problem.

Force the LLM to follow the instructions.

Figure 5: *Planning Agent*: Prompt for Plan Generation.

Coding Agent: Prompt for Plan Verification with Simulation

```
You are a programmer tasked with verifying a plan to solve a given problem using the
**{language}** programming language.

## Problem
{problem}

### Plan
{plan}

**Expected Output:**
Your response must be structured as follows:

### Simulation
- Take a sample input and apply plan step by step to get the output.
- Compare the generated output with the sample output to verify if
  your plan works as expected.

### Plan Evaluation
- If the simulation is successful write **No Need to Modify Plan**.
- Otherwise write **Plan Modification Needed**.
```

To validate a plan, a human programmer typically simulates it with sample input, generates the corresponding output, and compares the generated output to the expected result. At this stage, we instruct the LLM to perform this process as well, ensuring it follows the same steps like human programmer to confirm the validity of the plan.

Finally, tell the LLM to write done it's decision in a specific format.

Figure 6: *Planning Agent*: Prompt for Plan Verification with the help of Simulation.

Coding Agent: Prompt for Plan Refinement

```
You are a programmer tasked with generating appropriate plan to solve a given problem
using the **{language}** programming language. You already have a wrong plan.
Correct it so that it can generate correct code.

## Problem
{problem}

### Plan
{plan}

## Plan Critique

{plan_verifical_report_from_previous_step}

**Expected Output:**
Your response must be structured as follows:

### New Plan
- Write down a detailed, step-by-step modified plan to solve the **original problem**.
- Ensure each step logically follows from the previous one.

--------
**Important Instruction:**
- Your response must contain only the plan.
- Do not add any explanation.
- Do not generate code.
```

Provide all the details and instruct the LLM to generate revised plan.

Figure 7: *Planning Agent*: Prompt for Plan Refinement.

Coding Agent: Prompt for Code Generation

````
You are a programmer tasked with solving a given problem using the **{language}**
programming language. See the plan to solve the plan and implement code to solve it.

## Problem
{problem}

### Plan
{plan}

--------
**Important Instruction:**
- Do not add any explanation.
- The generated **{language}** code must be inside a triple backtick (```) code block.
````

Figure 8: *Coding Agent*: Prompt for Code Generation.

An Example from HumanEval dataset for demonstrating how CODESIM works

Input for Planning: 1
You are a programmer tasked with generating appropriate plan to solve a given problem using the Python3 programming language.

Problem

```
def generate_integers(a, b):
    """
    Given two positive integers a and b, return the even digits between a
    and b, in ascending order.

    For example:
    generate_integers(2, 8) => [2, 4, 6, 8]
    generate_integers(8, 2) => [2, 4, 6, 8]
    generate_integers(10, 14) => []
    """
```

Expected Output:

Your response must be structured as follows:

Debugging Agent: Prompt for Debugging

```
You are a programmer who has received a solution of a problem written in **{language}**
that fails to pass certain test cases. Your task is to modify the code in such a way so
that it can pass all the test cases. Do not generate same code.

## Problem
{problem}

### Plan
{plan}

### Buggy Code
{code}

### Test Report
{test_log}

**Expected Output:**
Your response must be structured as follows:

### Simulation with failed test case
To detect where is the bug:
- Take a sample test case where it fails.
- Take the input go through each step according to the plan
- You will get a output that must be different from the expected output.

### Debugging Notes
- Based on this simulation detect any of the following cases:
    - Plan is wrong
    - Plan is correct but plan to code generation is wrong.
- Finally, discuss how to correct this code.

### Modified Code
# Your corrected code, with comments explaining each correction.

--------
**Important Instruction:**
- Strictly follow the instructions.
- Do not add testing code for example assert statement in your code.
- Do not be overconfident that the generated code is correct. It is wrong.
- The modified **{language}** code must be enclosed within triple backticks (```).
- Your response must contain **Simulation with failed test case**, **Debugging Notes**,
  and **Modified Code** section.
```

To detect where a program fails, a human programmer typically selects a sample test case, simulates it line by line, and observes the output at each step. This process helps in identifying and debugging issues within the code. Similarly, we have instructed the LLM to select a test case where the code fails, simulate it, identify the bug through step-by-step analysis, and then debug the issue accordingly.

Based on the simulation output we have instruct the LLM to write down how to correct the code.

Finally, instruct the LLM to generate the modified code.

Figure 9: *Debugging Agent*: Prompt for Debugging.

Problem Understanding

- Think about the original problem. Develop an initial understanding about the problem.

Recall Example Problem

Recall a relevant and distinct problems (different from problem mentioned above) and
- Describe it
- Generate Python3 code step by step to solve that problem
- Discuss the algorithm to solve this problem
- Finally generate a planning to solve that problem

Algorithm to solve the original problem

- Write down the algorithm that is well suited for the original problem
- Give some tutorials to about the algorithm for example:

- How to approach this type of algorithm
- Important things to consider

Plan

- Write down a detailed, step-by-step plan to solve the original problem.

——

Important Instruction:
- Strictly follow the instructions.
- Do not generate code.

Response from Planning: 1

Problem Understanding
The problem requires generating a list of even integers between two given positive integers, a and b, inclusive. The list should be in ascending order, regardless of whether a is less than or greater than b. If there are no even integers in the range, the function should return an empty list.

Recall Example Problem
Example Problem:

Find all odd numbers between two given integers x and y, inclusive, and return them in descending order.

Step-by-step Solution:

1. Determine the range between x and y.
2. Identify all odd numbers within this range.
3. Sort the odd numbers in descending order.
4. Return the sorted list.

Algorithm:

1. Check if x is greater than y. If so, swap them to ensure the range is correct.
2. Iterate through the range from x to y.
3. For each number, check if it is odd (i.e., $number\%2! = 0$).
4. Collect all odd numbers in a list.
5. Sort the list in descending order.
6. Return the sorted list.

Planning:

- Define a function $find\_odd\_numbers(x, y)$.
- Use a conditional to swap x and y if $x > y$.
- Use a loop to iterate through the range from x to y.
- Check each number for oddness and add it to a list if it is odd.

- Sort the list in descending order.
- Return the list.

Algorithm to Solve the Original Problem

1. Determine the Range:
- Identify the start and end of the range by taking the minimum and maximum of a and b.

2. Iterate Through the Range:
- Loop through each number in the range from the minimum to the maximum of a and b.

3. Check for Even Numbers:
- For each number, check if it is even by using the condition $number\%2 == 0$.

4. Collect Even Numbers:
- Add each even number to a list.

5. Return the List:
- Since the numbers are collected in ascending order, return the list directly.

Tutorials:

- Approach:
- Always ensure the range is correctly defined, especially when the order of inputs can vary.
- Use modulus operation to check for evenness or oddness.

- Important Considerations:
- Handle edge cases where there might be no even numbers in the range.
- Ensure the function is robust to handle any positive integer inputs.

Plan

1. Define the Function:
- Create a function $generate\_integers(a, b)$.

2. Determine the Range:
- Use $min(a, b)$ and $max(a, b)$ to define the start and end of the range.

3. Initialize an Empty List:
- Prepare an empty list to store even numbers.

4. Loop Through the Range:
- Use a for loop to iterate from the start to the end of the range.

5. Check for Evenness:
- Within the loop, use $ifnumber\%2 == 0$ to check if a number is even.

6. Collect Even Numbers:
- Append even numbers to the list.

7. Return the Result:
- Return the list of even numbers, which will be in ascending order.

By following this plan, the function will correctly generate a list of even integers between the given numbers a and b.

Input for Simulation: 1
You are a programmer tasked with verifying a plan to solve a given problem using the Python3 programming language.

Problem:

```
def generate_integers(a, b):
    """
    Given two positive integers a and b, return the even digits between a
    and b, in ascending order.

    For example:
    generate_integers(2, 8) => [2, 4, 6, 8]
    generate_integers(8, 2) => [2, 4, 6, 8]
    generate_integers(10, 14) => []
    """
```

Plan

1. Define the Function:
- Create a function $generate\_integers(a, b)$.

2. Determine the Range:
- Use $min(a, b)$ and $max(a, b)$ to define the start and end of the range.v
3. Initialize an Empty List:
- Prepare an empty list to store even numbers.

4. Loop Through the Range:
- Use a for loop to iterate from the start to the end of the range.

5. Check for Evenness:
- Within the loop, use $ifnumber\%2 == 0$ to check if a number is even.

6. Collect Even Numbers:
- Append even numbers to the list.

7. Return the Result:
- Return the list of even numbers, which will be in ascending order.

By following this plan, the function will correctly generate a list of even integers between the given numbers a and b.

Expected Output:

Your response must be structured as follows:

Simulation

- Take a sample input and apply plan step by step to get the output.
- Compare the generated output with the sample output to verify if your plan works as expected.

Plan Evaluation

- If the simulation is successful write No Need to Modify Plan.
- Otherwise write Plan Modification Needed.

Response from Simulation: 1
Simulation

Let's apply the plan step by step using the sample input $generate\_integers(2, 8)$.

1. Define the Function:
- We define the function $generate\_integers(a, b)$.

2. Determine the Range:
- Calculate $start = min(2, 8) = 2$ and $end = max(2, 8) = 8$.

3. Initialize an Empty List:
- Create an empty list $even_numbers = []$.

4. Loop Through the Range:
- Iterate over the range from $start$ to end (inclusive).

5. Check for Evenness:
- For each number in the range, check if it is even using $if number\%2 == 0$.

6. Collect Even Numbers:
- Append even numbers to the list $even_numbers$.

7. Return the Result:
- Return the list $even_numbers$.

Applying these steps:

- Range: 2 to 8
- Iteration:
- 2 is even, append to list: $[2]$
- 3 is not even
- 4 is even, append to list: $[2, 4]$
- 5 is not even

- 6 is even, append to list: $[2, 4, 6]$
- 7 is not even
- 8 is even, append to list: $[2, 4, 6, 8]$

The output is $[2, 4, 6, 8]$, which matches the expected output.

Plan Evaluation

No Need to Modify Plan

The plan works as expected for the given problem. The function correctly generates a list of even integers between the given numbers a and b, regardless of their order.

Input for final code generation:

You are a programmer tasked with solving a given problem using the Python3 programming language. See the plan to solve the plan and implement code to solve it.

Problem:

```
def generate_integers(a, b):
    """
    Given two positive integers a and b, return the even digits between a
    and b, in ascending order.

    For example:
    generate_integers(2, 8) => [2, 4, 6, 8]
    generate_integers(8, 2) => [2, 4, 6, 8]
    generate_integers(10, 14) => []
    """
```

Plan

1. Define the Function:
- Create a function $generate\_integers(a, b)$.

2. Determine the Range:
- Use $min(a, b)$ and $max(a, b)$ to define the start and end of the range.

3. Initialize an Empty List:
- Prepare an empty list to store even numbers.

4. Loop Through the Range:
- Use a for loop to iterate from the start to the end of the range.

5. Check for Evenness:
- Within the loop, use $if number \% 2 == 0$ to check if a number is even.

6. Collect Even Numbers:
- Append even numbers to the list.

7. Return the Result:
- Return the list of even numbers, which will be in ascending order.

By following this plan, the function will correctly generate a list of even integers between the given numbers a and b.

———

Important Instructions:
- Do not add any explanation.
- The generated Python3 code must be inside a triple backtick ($$$) code block.

Response from final code generation:

```
def generate_integers(a, b):
    start = min(a, b)
    end = max(a, b)
    even_numbers = []
    for number in range(start, end + 1):
        if number % 2 == 0:
            even_numbers.append(number)
    return even_numbers
```

Input for Improving code: 1, 1

You are a programmer who has received a solution of a problem written in Python3 that fails to pass certain test cases. Your task is to modify the code in such a way so that it can pass all the test cases. Do not generate same code.

Problem:

```
def generate_integers(a, b):
    """
    Given two positive integers a and b, return the even digits between a
    and b, in ascending order.

    For example:
    generate_integers(2, 8) => [2, 4, 6, 8]
    generate_integers(8, 2) => [2, 4, 6, 8]
    generate_integers(10, 14) => []
    """
```

Plan

1. Define the Function:
- Create a function $generate\_integers(a, b)$.

2. Determine the Range:
- Use $min(a, b)$ and $max(a, b)$ to define the start and end of the range.

3. Initialize an Empty List:
- Prepare an empty list to store even numbers.

4. Loop Through the Range:
- Use a for loop to iterate from the start to the end of the range.

5. Check for Evenness:
- Within the loop, use $ifnumber\%2 == 0$ to check if a number is even.

6. Collect Even Numbers:
- Append even numbers to the list.

7. Return the Result:
- Return the list of even numbers, which will be in ascending order.

By following this plan, the function will correctly generate a list of even integers between the given numbers a and b.

Buggy Code

```
def generate_integers(a, b):
    start = min(a, b)
    end = max(a, b)
    even_numbers = []
    for number in range(start, end + 1):
        if number % 2 == 0:
            even_numbers.append(number)
    return even_numbers
```

Test Report

Test Cases where the generated code failed to generate the expected output:
$assertgenerate_integers(10, 14) == []$

Expected Output:

Your response must be structured as follows:

Simulation with failed test case
To detect where is the bug:
- Take a sample test case where it fails.
- Take the input go through each step according to the plan
- You will get a output that must be different from the expected output.

Debugging Notes
Based on this simulation detect any of the following cases:
- Plan is wrong
- Plan is correct but plan to code generation is wrong.

- Finally, discuss how to correct this code.

Modified Code

"'Python3
Your corrected code, with comments explaining each correction.
"'

———

Important Instructions:
- Strictly follow the instructions.
- Do not add testing code for example assert statement in your code.
- Do not be overconfident that the generated code is correct. It is wrong.
- The modified Python3 code must be enclosed within triple backticks ("').
- Your response must contain Simulation with failed test case, Debugging Notes, and Modified Code section.

Response from Improving code: 1, 1

Simulation with failed test case

Let's take the test case $generate\_integers(10, 14)$:v
1. Determine the Range:
- $start = min(10, 14) = 10$
- $end = max(10, 14) = 14$

2. Loop Through the Range:
- The loop iterates over $10, 11, 12, 13, 14$.

3. Check for Evenness:
- $10\%2 == 0$ (even, added to list)
- $11\%2! = 0$ (odd, not added)
- $12\%2 == 0$ (even, added to list)
- $13\%2! = 0$ (odd, not added)
- $14\%2 == 0$ (even, added to list)

4. Collect Even Numbers:
- The list becomes $[10, 12, 14]$.

5. Return the Result:
- The function returns $[10, 12, 14]$.

Expected Output: []

Debugging Notes

- Plan is correct: The plan correctly identifies the need to find even numbers between a and b.
- Plan to code generation is wrong: The code does not account for the requirement that the even digits themselves must be between a and b. It incorrectly includes numbers like $10, 12, 14$ when the digits themselves (1, 2, 4) are not between 10 and 14.

Modified Code

```
def generate_integers(a, b):
    start = min(a, b)
    end = max(a, b)
    even_numbers = []
    for number in range(start, end + 1):
        # Check each digit in the number
        for digit in str(number):
            if int(digit) % 2 == 0 and start <= int(digit) <= end:
                even_numbers.append(int(digit))
    # Remove duplicates and sort the result
    return sorted(set(even_numbers))
```

Explanation of Corrections:

- Digit Check: The code now checks each digit of the numbers in the range to see if it is even and within the specified range.
- Set for Uniqueness: A set is used to ensure that each even digit is only included once.
- Sorting: The final list is sorted to ensure ascending order.