Cogito, ergo sum: A Neurobiologically-Inspired Cognition-Memory-Growth System for Code Generation

Anonymous ACL submission

Abstract

Large language models-based Multi-Agent Systems (MAS) have demonstrated promising performance for enhancing the efficiency and accuracy of code generation tasks. However, most existing methods follow a conventional sequence of planning, coding, and debugging, which contradicts the growth-driven nature of human learning process. Additionally, the frequent information interaction between multiple agents inevitably involves high computational costs. In this paper, we propose **Cogito**, a neurobiologically-inspired multi-agent framework to enhance the problem-solving capabilities in code generation tasks with lower cost. Specifically, **Cogito** adopts a reverse sequence: it first undergoes debugging, then coding, and finally planning. This approach mimics human learning and development, where knowledge is acquired progressively. Accordingly, a hippocampus-like memory module with different functions is designed to work with the pipeline to provide quick retrieval in similar tasks. Through this growth-based learning model, **Cogito** accumulates knowledge and cognitive skills at each stage, ultimately forming a Super-Role—an all-capable agent to perform the code generation task. Extensive experiments against representative baselines demonstrate the superior performance and efficiency of **Cogito**. The code is publicly available at anonymous.4open.science/r/test_80EF.

1 Introduction

Large language models (LLMs) have demonstrated remarkable capabilities in code generation (Chowdhery et al., 2022), testing (Fakhoury et al., 2024), and debugging (Xia and Zhang, 2023). Recent advances highlight the effectiveness multi-agent collaborative, surpassing single-agent approaches in software development tasks (Islam et al., 2024; Rasheed et al., 2024a). These advancements not only automate complex programming workflows but also enhance the models' reasoning and problem-solving abilities, attracting attention from both academia and industry institutions like OpenAI [1] and Meta AI [2].

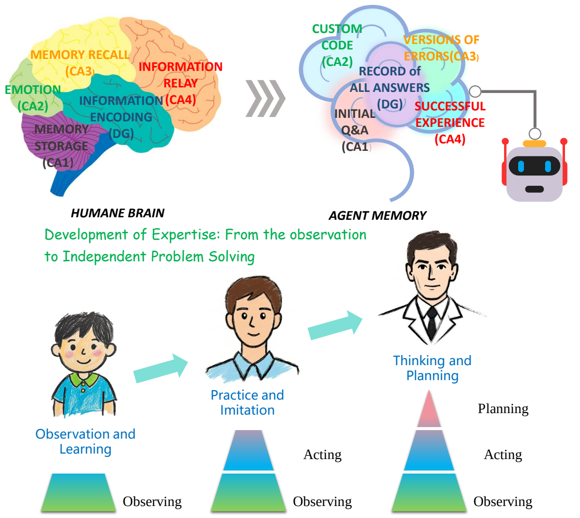


Figure 1: The intuitions behind this work. (Top): brain's different regions are dedicated to distinct functions and tasks. Inspired by this functional specialization, we design an agent with distinct roles that evolve through stages. (Bottom): the growth trajectory of an individual, progressing from observation and learning in childhood, to practice and imitation in young adulthood, and finally to independent problem-solving and planning in the expert stage.

Despite these achievements, existing frameworks rigidly follow a "plan-first" sequence: agents plan, code, then debug (Islam et al., 2024, 2025). While this mimics traditional software workflows, it fundamentally conflicts with human learning principles. For example, humans learn through trial and error (debugging) before developing systematic knowledge (planning), a process supported by cognitive theories like productive failure (Figure 1).

[1] https://api.semanticscholar.org/CorpusID:257532815
[2] https://api.semanticscholar.org/CorpusID:271571434

However, these methods lead to two critical flaws. First, agents repeatedly solve similar bugs without learning from past mistakes, mirroring a "cognitive misalignment" between machines and humans. Second, frequent inter-agent communication drastically increases computational costs (Huang et al., 2023). These limitations highlight the need for frameworks that align with human-like learning while reducing overhead.

Inspired by neurobiological principles, We propose **Cogito**, a multi-agent framework that flips the traditional workflow to debug→code→plan, directly inspired by human cognitive growth. Instead of forcing agents to plan before acting, Cogito lets them learn from failures first, akin to how children develop skills. At its core, Cogito integrates a hippocampus-like memory module for structured storage and retrieval: short-term memory captures debugging experiences (e.g., error patterns and fixes), while long-term storage retains validated solutions for future reuse. Over time, specialized agents merge into a unified Super-Role that internalizes collective expertise, eliminating constant communication. This dynamic learning process transforms code generation from a rigid pipeline into an evolving practice where agents "grow smarter" through experience, bridging the gap between artificial and human intelligence. **The key contributions are summarized as follows**:

- We propose **Cogito**, the first framework that enables a **Super-Role** agent to progressively evolve through a human-inspired debug→code→plan sequence, whcih is contrary to conventional workflows. This biologically grounded approach is akin to human expertise development process.

- We design a hippocampus-inspired memory that stores different content based on learning stages, where different parts of memory are inter-connected to ensure the completeness of stored information. The design can support dynamic and adaptive programming workflows.

- We conduct extensive experiments to validate **Cogito**'s efficiency on eight code generation tasks. The results show that **Cogito** reduces token consumption by up to 66.29% and improves performance by an average of 12.2% compared to MapCoder, using GPT-3.5-turbo and GPT-4.

2 Related work

2.1 LLM Agents

LLM-based agents normally consist of four core components: planning, memory, perception, and action. Planning and memory form the cognitive core, while perception and action enable interaction with the environment to achieve goals (Xi et al., 2023). The planning component decomposes complex tasks into manageable subtasks and schedules their execution to achieve predefined objectives, while also incorporating the flexibility to adapt plans dynamically in response to external feedback. The memory component, on the other hand, stores historical actions and observations, enabling agents to draw on past experiences to refine decision-making processes and enhance task execution efficiency. This dual approach facilitates continuous learning and optimization, ensuring improved performance over time. Effective memory management is critical for system performance (Wang et al., 2023; Zhang et al., 2024b). Due to the suitability of this setup for code generation problems, a large number of works have emerged in this field (Wang et al., 2024; Liu et al., 2024).

2.2 Multi-Agent Collaboration for Software Development

To effectively solve complex problems, tasks are divided into specialized roles, each handling a specific aspect of the process. This role-based division, combined with agent collaboration, boosts efficiency and enhances outcomes. The typical workflow includes task refinement, execution, result validation, and optimization (Lei et al., 2024b,a). These stages ensure that each component is managed with focus, leading to smoother task execution and more reliable results. For instance, MetaGPT (Hong et al., 2023) mimics standardized real-world collaboration procedures, incorporating five distinct roles. Similarly, MapCoder (Islam et al., 2024) adapts the human programming cycle to define four key roles for task completion, while CodeSim (Islam et al., 2025) further advances this idea by enhancing code generation through human-like planning, coding, and debugging with step-by-step input/output simulation.

2.3 Prompt Engineering

Prompt engineering plays a crucial role in optimizing code generation tasks by effectively guiding

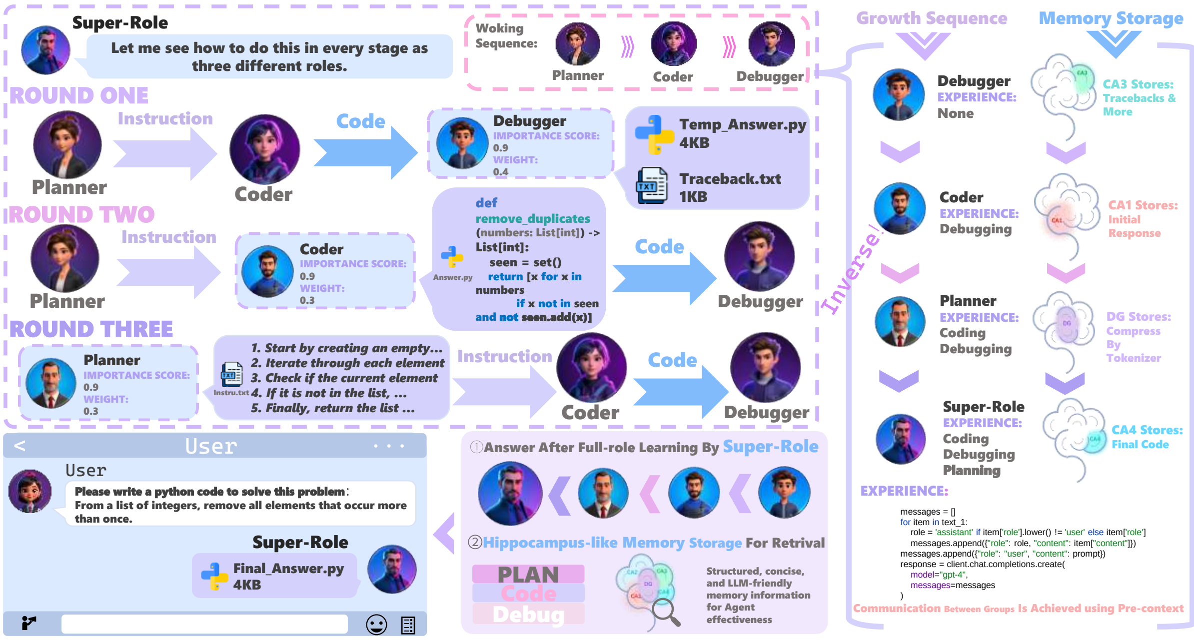


Figure 2: Overview of **Cogito**. The upper section illustrates the learning process of the **Super-Role** stored in the memory module. The lower section provides a detailed explanation of the process: initially, it assumes the role of the debugger within the group, followed by transitions to the coder and planner roles. After completing the learning cycle, the final answer is provided by the **Super-Role**.

model outputs, ensuring both, consistency and efficiency in the process. Inspired by the CoT (Chain of Thought) (Wei et al., 2022) method, there are usually three main stages in code generation tasks to gradually solve the problem while maintaining clarity and structured reasoning: Planning (Talebirad and Nadiri, 2023; Zhang et al., 2024a; Lin et al., 2024), Coding (Rasheed et al., 2024a; Zan et al., 2024; Tao et al., 2024), and Debugging (Li et al., 2023; Qin et al., 2024; Rasheed et al., 2024b), AgentCoder (Huang et al., 2023) directs the agent to produce pseudocode following the phases of problem comprehension and algorithm selection. LLM4CBI (Tu et al., 2023) utilizes a stored component that tracks relevant prompts and selects the most effective ones to guide LLMs in generating variations.

3 Cogito

3.1 Agent Roles

Building on the "Chain of Thought" (CoT) (Wei et al., 2022) process, we assign three distinct roles within the team: Planner, Coder, and Debugger. The Planner's role is to outline a clear, step-by-step strategy for solving the problem, considering key aspects such as edge cases and performance issues. This guidance helps the Coder translate the plan into functional code, ensuring that all critical scenarios are addressed during implementation. After the Coder finishes coding, the solution is tested against a set of sample inputs and expected outputs (Islam et al., 2024). If the code passes the tests, it is considered finalized. However, if it fails, the Debugger steps in, analyzing the traceback feedback to identify and correct errors. This collaborative process ensures that the final code is both robust and efficient.

3.2 Super-Role

In this experimental setup, we introduce a shared member known as the **Super-Role**, who is assigned to each of the three groups sequentially. This member rotates through the roles of Debugger, Coder, and Planner within each group, contributing to a dynamic and collaborative environment. Importantly, the public role retains the memory of all its prior experiences, which plays a crucial role in informing and guiding the execution of its current responsibilities. This memory not only enhances the efficiency of the member's actions within the same group but also acts as a communication bridge across different groups, facilitating the transfer of knowledge and strategies. Upon completion of the three distinct tasks, the public member now equipped with accumulated expertise will be entrusted with the task of solving the problem independently. To ensure robustness in the final solution, the member is provided with up to five opportunities for error correction, allowing iterative refinement of the outcome. The final answer, enriched by the cumulative knowledge gained through this process, will be generated and presented by the **Super-Role**, reflecting its comprehensive learning journey across multiple roles and tasks. A complete example of the response process is shown in Figure 3.

3.3 The Hippocampus-like Memory Module

Inspired by the structural divisions of the hippocampus (Burgess et al., 2002; Berron et al., 2017; Kesner, 2013), we design a memory-enhanced storage module that aligns with its specialized functions. The hippocampus, including regions like CA1–CA4 and the Dentate Gyrus (DG), encodes, consolidates, and retrieves memories. Our model maps task-specific information and generated code to different regions, mirroring hierarchical and associative memory mechanisms. This biologically inspired design enhances long-term retention, contextual recall, and adaptive retrieval, optimizing memory utilization for efficient code generation.

DG Part. The **Dentate Gyrus (DG)** plays a key role in memory formation by performing pattern separation, transforming similar inputs into distinct, interference-resistant representations. Inspired by this, our memory module integrates a tokenizer to process diverse task information, organizing it into distinct memory traces to enhance retrieval efficiency and adaptability in code generation.

CA1 Region. The **CA1** region, pivotal for the storage and retrieval of long-term memories, serves as the repository for initial responses generated during problem-solving. Once formulated, these responses are stored for long-term access and ready for future retrieval when related tasks arise, much like how we retain foundational knowledge and lessons learned over time.

CA2 Region. While research on the **CA2** region remains sparse, its role in social and emotional memory inspired our "Personalization Module." Here, users can input prior code, enabling the system to learn their naming conventions and coding style. While optional, this module enhances alignment with user preferences, fostering a more intuitive coding experience.

CA3 Region. The **CA3** region, which facilitates quick recall and rapid learning, stores different versions of the code along with the associated error tracebacks. This allows for fast retrieval of past

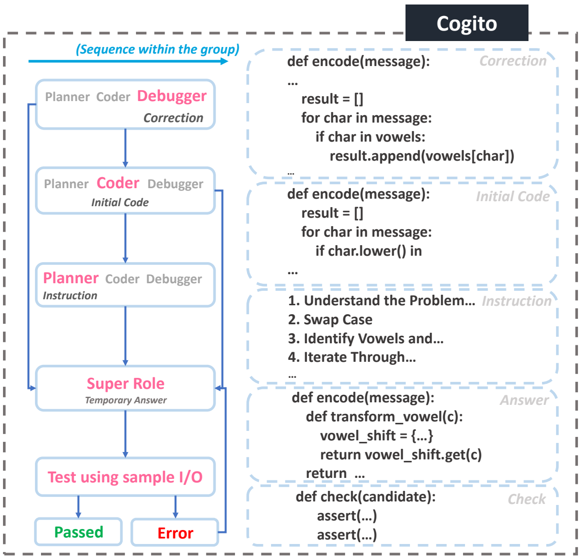


Figure 3: The abbreviated explanation of the process and sample outputs for each step.

mistakes and corrections, helping avoid errors in future problem-solving processes. This mirrors the brain's ability to learn from past experiences, making future decision-making faster and more efficient.

CA4 Region. Finally, the **CA4** region serves as a bridge between the **DG** and **CA3**, storing only the final, correct result or the last modified version. This ensures that successful outcomes are quickly accessible for similar tasks, enabling efficient problem-solving and minimizing the time spent on recurring issues.

3.4 Agent Collaboration Settings

To mitigate the impact of low-quality answers during learning, we assign initial weights of 0.4, 0.4, and 0.3 to each role. Importance scores are performance-based: 0.9 if the generated code passes tests in debugging and coding, otherwise 0.1, while planning consistently receives 0.9. The final score aggregates across stages, emphasizing high-quality outputs. To enhance variability and prevent over-reliance on faulty answers, two newly selected roles—excluding the **Super-Role**—are reintroduced in each group. Algorithm 1 summarizes our agent traversal.

4 EXPERIMENTS

4.1 Experimental Settings

Datasets. We adopt 8 widely-used benchmark datasets for testing, with 5 datasets contain-

Algorithm 1 Cogito

Common Agent: A_c, **Plan Agent:** A_p
Implement Agent: A_i, **Debug Agent:** A_d
Super Role: S_R
Plan_A $\leftarrow A_p$(Question)
Code_A $\leftarrow A_i$(Plan_A, Question)
Own_Answer $\leftarrow A_c$(Code_A, sample_io)
Plan_B $\leftarrow A_p$(Question)
Own_Code $\leftarrow A_i$(Plan_B, Question)
Answer_B $\leftarrow A_d$(Code_B, sample_io)
Own_Plan $\leftarrow A_p$(Question)
Code_C $\leftarrow A_i$(Plan_C, Question)
Answer_C $\leftarrow A_d$(Code_C, sample_io)
tem_code $\leftarrow S_R$(Question, **Own**_Answer, **Own**_Code, **Own**_Plan)
if test(tem_code, sample_io) **then**
    return tem_code
else
    for $i = 1$ **to** 5 **do**
        code $\leftarrow S_R$(Question, **Own**_Answer, **Own**_Code, **Own**_Plan)
        if test(code, sample_io) **then**
            return code
        end if
        tem_code $\leftarrow$ code
    end for
    return tem_code
end if

ing only simple programming problems (e.g., HumanEval (Chen et al., 2021), HumanEval-ET (Dong et al., 2023a), EvalPlus (Liu et al., 2023), MBPP (Austin et al., 2021), MBPP-ET (Dong et al., 2023a)), and others that contain complex programming problems (e.g., Automated Programming Progress Standard (APPS), xCodeEval (Khan et al., 2023), and CodeContest). More detailed information is provided in Appendix B.1.

Evaluation Metric. For the dataset used in the experiment, we uniformly apply the widely-used unbiased version of **Pass @k** as evaluation metric (Chen et al., 2021; Dong et al., 2023b). Note that the unbiased version of **Pass @k** is a metric used to evaluate recommendation systems by correcting for potential biases in the recommendation process. The formula is given by:

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

where n is the total number of items, c is the number of relevant items, and k is the size of the top-k recommendations. More detailed information is provided in Appendix B.2.

Baselines. We conduct a comprehensive comparison with several representative methods: Direct, Chain-of-Thought (CoT), Self-Planning, Analogical Reasoning, MapCoder, and CodeSim. More detailed information is provided in Appendix B.3.

4.2 Overall Performance

In this section, we conduct a comprehensive evaluation of our proposed process, and all the results are systematically presented in Table 1. From the table, it is evident that **Cogito** outperforms all the other models, achieving the highest scores across all datasets. Notably, the application of GPT-4 significantly enhances the overall performance, yielding the best results observed in our experiments. These results underscore the effectiveness of **Cogito** and highlight the substantial improvements brought by the integration of GPT-4 into the process, demonstrating its potential for high-level performance across diverse data scenarios. The results further validate the effectiveness of our growth-based learning concept, demonstrating that enabling the agent to evolve reversely can enhance its problem-solving capabilities.

4.2.1 Performance on Simple Code Generation Tasks

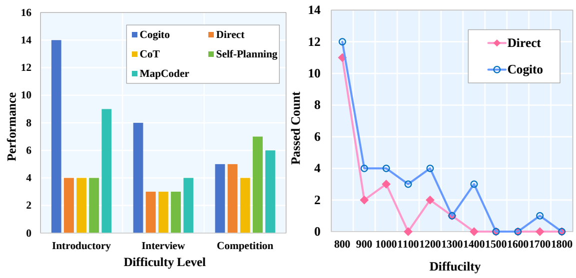
Table 1 summarizes the performance of various baselines and the average percentage gains achieved by our method. Compared to the state-of-the-art CodeSim, Cogito yields notable Pass@1 improvements of 4.88%, 13.47%, and 12.43% on HumanEval, HumanEval-ET, and EvalPlus, respectively, using GPT-3.5-turbo. Against direct prompting, Cogito achieves up to a 119.65% gain. Leveraging GPT-4 further enhances performance across all datasets, achieving the highest scores in our experiments. Additionally, performance remains stable even with more test cases per problem, highlighting the robustness of Cogito's code and its ability to handle edge cases. These results collectively demonstrate Cogito's strong generalization and reliability across diverse evaluation settings.

4.2.2 Performance on Complex Code Generation Tasks

Contest-level problems feature more comprehensive problem descriptions and a greater number of test cases, with no limitation on the generation of

LLM	Approach	Simple Problems					Contest-Level Problems		
		HumanEval	HumanEvalET	EvalPlus	MBPP	MBPPET	APPS	xCodeEval	CodeContest
ChatGPT-3.5	Direct †	48.1	37.2	66.5	49.8	37.7	8.0	17.9	5.5
	CoT †	68.9	55.5	65.2	54.5	39.6	7.3	23.6	6.1
	Self-Planning †	60.3	46.2	-	55.7	41.9	9.3	18.9	6.1
	Analogical †	63.4	50.6	59.1	70.5	46.1	6.7	15.1	7.3
	Reflexion †	67.1	49.4	62.2	73.0	47.4	-	-	-
	MapCoder †	80.5	70.1	71.3	78.3	54.4	11.3	27.4	12.7
	CodeSim ∗	86.0	72.0	73.2	86.4	59.7	12.0	-	16.4
	Cogito (Ours)	**90.2** ↑ **37.4%**	**81.7** ↑ **57.2%**	**82.3** ↑ **24.9%**	**85.1** ↑ **32.1%**	**59.7** ↑ **31.0%**	**18.0** ↑ **107.2%**	**30.2** ↑ **53.3%**	**13.3** ↑ **74.7%**
GPT-4	Direct †	80.1	73.8	81.7	81.1	54.7	12.7	32.1	12.1
	CoT †	89.0	61.6	-	82.4	56.2	11.3	36.8	5.5
	Self-Planning †	85.4	62.2	-	75.8	50.4	14.7	34.0	10.9
	Analogical †	66.5	48.8	62.2	58.4	40.3	12.0	26.4	10.9
	Reflexion †	91.0	78.7	81.7	78.3	51.9	-	-	-
	MapCoder †	93.9	82.9	83.5	83.1	57.7	22.0	45.3	28.5
	CodeSim ∗	94.5	81.7	84.8	89.7	61.5	22.0	-	29.1
	Cogito (Ours)	**95.7** ↑ **13.1%**	**83.5** ↑ **23.4%**	**85.4** ↑ **9.8%**	**88.2** ↑ **14.3%**	**66.3** ↑ **26.4%**	**27.3** ↑ **86.3%**	**47.2** ↑ **39.4%**	**29.7** ↑ **156.1%**

Table 1: Overall performance comparison across various datasets, categorized into *Simple Problems* and *Contest-Level Problems*. Cogito's performance is highlighted in **blue**. The average improvement is highlighted in **red**. †: Results are publicly disclosed in the paper of MapCoder. ∗: Results are publicly disclosed in the paper CodeSim.



(a) (b)

Figure 4: The comparison results on representative datasets.

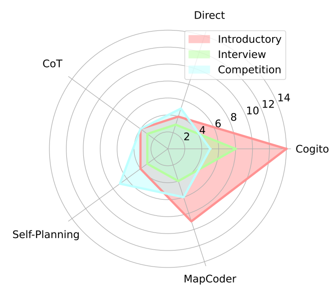


Figure 5: The comparison results with respect to the algorithm and difficulty levels (APPS dataset).

a single function to address the task. Cogito has shown notable advancements compared to multiple methods across most datasets, including APPS, xCodeEval, and CodeContests. Specifically, When using GPT-4, our approach achieves improvements of 24.09% and 2.06% over CodeSim, respectively. Adhering to the unified testing validation approach, we require that all responses within these three datasets be implemented as functions that take a string parameter, returning the result as a string via the 'return' statement. Despite the advantages of this methodology, its application has led to a decline in performance on certain platforms, notably xCodeEval and CodeContests.

4.2.3 Performance Under Different Difficulty Levels

Difficulty levels. The APPS dataset consists of problems with three difficulty levels: (i) Introductory, (ii) Interview, and (iii) Competition. Figure 4(a) and Figure 5 show the number of problems solved by different methods at different levels under these three classifications. At the Introductory and Interview levels, Cogito significantly outperforms existing methods, highlighting the effectiveness of our approach for relatively simple and moderately difficult code generation tasks.

Difficulty score. In the xCodeEval dataset, each task is assigned a difficulty score, with the difficulty scores of the answers that successfully pass the tests ranging from 800 to 1800. We compare our approach with the direct method, and our results consistently outperforme the direct method across

LLM	Dataset	Average for Cogito		Average for MapCoder		Average for CodeSim		Average API Calls Reduction	Average Token Reduction(k)	Average ACC Gain
		API Calls	Tokens (k)	API Calls	Tokens (k)	API Calls	Tokens (k)			
ChatGPT-3.5	HumanEval	10	6.26	17	10.41	7	5.48	2	1.70	6.95%
	MBPP	9	4.60	12	4.84	6	4.24	0	-0.06	10.6%
	APPS	14	14.74	21	26.57	15	19.20	4	8.15	6.35%
	xCodeEval	16	18.90	19	24.10	-	-	3	5.20	2.8%
	CodeContest	15	28.56	23	34.95	16	24.02	4.5	0.93	-1.25%
GPT-4	HumanEval	10	7.10	15	12.75	5	5.15	0	1.85	1.5%
	MBPP	10	4.80	8	4.96	5	5.21	-3.5	0.29	1.8%
	APPS	13	21.96	19	31.80	13	23.18	3	5.53	5.3%
	xCodeEval	14	17.93	14	23.45	-	-	0	5.52	1.9%
	CodeContest	15	32.35	19	38.70	17	41.66	3	7.67	0.9%
Average		**13**	**16.0**	16.7	21.25	9.3	13.64	↓ **1.6**	↓ **3.68**	

Table 2: The number of API calls and token consumption for different tasks, compared to the usage reduction with MapCoder and CodeSim.

Model	Pass@1	Perforcemance Drop
Cogito w/o Planning Experience	76.22	14.02
Cogito w/o Implementation Experience	78.05	12.19
Cogito w/o Debugging Experience	79.88	10.36
Cogito w/o Super-Role	69.33	20.91
Normal Sequence	73.17	17.07

Table 3: Ablation study results on HumanEval using GPT-3.5-turbo. The table shows the impact of different components or configurations on performance.

different difficulty levels (Figure 4(b)).

4.2.4 Consumption of API and Tokens

Table 2 reports the API calls and token usage (in thousands) of GPT-3.5-turbo and GPT-4 across various datasets. Compared to MapCoder on the HumanEval dataset, our method achieves up to 66.29% fewer tokens and 70% fewer API calls. On average, token consumption and API calls are reduced by 3.61% and 1.6%, respectively, across all tasks. While the complete developmental process introduces slight additional costs in certain scenarios—primarily due to multi-stage reasoning and role-based iteration—it consistently leads to higher pass rates by capturing broader contextual and structural nuances. Notably, GPT-4 exhibits higher resource consumption than GPT-3.5-turbo, which can be attributed to its tendency to generate longer and more detailed responses.

4.3 Ablation Study

Impact of Different Agents. To verify the effectiveness of the proposed approach, we systematically remove the active participation of various key roles involved in the process. The experimental results (Table 3) indicate that omitting the critical planning phase led to a maximum performance drop of 14.02%. The absence of hands-on Implementation practice reduces performance by 12.19%, while the lack of expert Debugging knowledge causes a 10.36% decline.

Impact of Work Sequence. To rigorously assess the distinctions between our work and previous approaches, particularly in terms of the sequence of experience accumulation, we conduct experiments in which we systematically alter the order in which experiences accumulate. Initially, we employ a sequence where the planner is introduced first, followed by the coder, and finally the debugger. Upon analyzing the results (Table 3), a performance drop of 17.07% clearly indicates that the altered sequence contributes to a significant deterioration in the final outcome.

Impact of Super-Role. In the third group, common roles are actively involved in every stage of the process. However, does this justify relying on them for the final answer? Table 3 presents the results. Even with extensive experience and insightful recommendations, planners may still fail to achieve desired outcomes due to a lack of hands-on experience or expertise in addressing practical challenges during implementation. Thus, the **Super-Role**'s final answer is indispensable.

Impact of Sample I/O. In this study, we augment the HumanEval dataset with input-output pairs from MapCoder (Islam et al., 2024) dataset and five additional test cases from HumanEval-ET dataset. Our results show a modest 0.6% improvement, indicating a slight positive effect on model

Dataset	Debug Times(t)	
	3	5
HumanEval	86.59	90.24
HumanEval-ET	78.05	81.71

Table 4: Analysis of debugging times on representative datasets.

Number of Groups	Results	
	Pass@1	Average Token(k)
Three Groups	90.24	10.41
Six Groups	80.49	17.43

Table 5: Analysis of debugging times on representative datasets.

performance. These findings suggest that dataset augmentation with diverse test cases can improve model accuracy.

4.4 Hyper-parameter Analysis

Impact of *t*. It involves a single hyperparameter: the number of self-debugging attempts, denoted as t. As shown in Table 4, increasing the value of t improves the performance. However, this enhancement comes with a trade-off, as it requires more computational time and an increased number of tokens to complete the process. This observation highlights the inherent balance between performance and resource consumption in the proposed method.

Impact of the Number of Iterations for Accumulating Experience. Initially, we set the number of roles to 3, indicating that we require three groups per experience-learning cycle. Increasing the number to six seems intuitive for better experience accumulation. However, as a detailed comparison provided in Table 5, increasing the number actually leads to decreased performance and higher token consumption. Such results indicate that setting one experience-learning cycle for Cogito is enough and reasonable for improving performance.

4.5 Case Study

4.5.1 New Random Roles vs. Same Roles

In group discussion sessions, each group will reintroduce two new random members to assume two distinct roles. The question arises: why not allow the same two members to continue participating

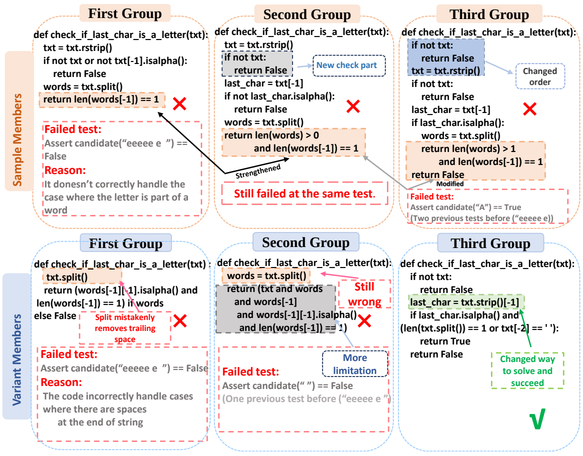


Figure 6: An example of answers from the same group members and different group members on HumanEval.

throughout the role transitions? The rationale behind this decision lies in the potential pitfalls of starting with an incorrect approach. If the direction of code writing is flawed in the initial phase, any subsequent improvements or redesigns will be built upon this foundational error. No matter how many times debugging is performed, the final result will inevitably remain compromised. Therefore, it is essential to ensure that the foundation is correct before moving forward with further development. An example is shown in Figure 6, demonstrating the necessity of introducing new random roles.

5 Conclusion

In this work, we introduce **Cogito**, a neurobiologically-inspired multi-agent framework for code generation that redefines the traditional workflow of planning, coding, and debugging by adopting a reverse approach. By mimicking the human growth process, **Cogito** progressively develops its capabilities, transitioning through specialized roles—Debugger, Coder, and Planner—and ultimately evolving into a **Super-Role** capable of autonomously handling complex code generation tasks. Through extensive evaluations on multiple representative datasets, **Cogito** demonstrates its ability to achieve state-of-the-art performance with higher efficiency and lower computational cost compared to existing methods. These results highlight the potential of biologically-inspired design principles in advancing intelligent systems. Future work will focus on further optimizing **Cogito**'s architecture and exploring its applicability to broader software engineering tasks.

Limitations

One limitation arises from the absence of the official evaluation mechanism when handling complex programming tasks. As a result, the traceback signals obtained from such problems may be inaccurate, which in turn hinders the model's ability to correctly identify and fix errors. To ensure compatibility with custom-written evaluation scripts, we further constrained the model to generate solutions in the form of a single function. While this simplification facilitates evaluation, it also reduces the expressiveness and potential correctness of the generated solutions.

Another challenge lies in the retrieval process. Due to the limited number of stored examples in memory and the lack of a dedicated retrieval database tailored for programming problems, the quality of the retrieved reference examples is often suboptimal. This negatively impacts the relevance and usefulness of retrieved contexts in guiding the model's generation.

References

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. 2021. Program synthesis with large language models. *ArXiv*, abs/2108.07732.

D. Berron, P. Vieweg, A. Hochkeppler, J.B. Pluta, S.-L. Ding, A. Maass, A. Luther, L. Xie, S.R. Das, D.A. Wolk, T. Wolbers, P.A. Yushkevich, E. Düzel, and L.E.M. Wisse. 2017. A protocol for manual segmentation of medial temporal lobe subregions in 7tesla mri. *NeuroImage: Clinical*, 15:466–482.

Neil Burgess, Eleanor A Maguire, and John O'Keefe. 2002. The human hippocampus and spatial and episodic memory. *Neuron*, 35(4):625–641.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, 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, and 34 others. 2021. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam M. Shazeer, Vinodkumar Prabhakaran, and 48 others. 2022. Palm: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311.

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

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023b. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*.

Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering*, 50:2254–2268.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zi Hen Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2023. Metagpt: Meta programming for a multi-agent collaborative framework.

Dong Huang, Jie M.Zhang, Michael Luck, Qi Bu, Yuhao Qing, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md. Rizwan Parvez. 2025. Codesim: Multi-agent code generation and problem solving through simulation-driven planning and debugging. *ArXiv*, abs/2502.05664.

Raymond P. Kesner. 2013. A process analysis of the ca3 subregion of the hippocampus. *Frontiers in Cellular Neuroscience*, 7.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md. Rizwan Parvez, and Shafiq R. Joty. 2023. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Annual Meeting of the Association for Computational Linguistics*.

Bin Lei, Yuchen Li, and Qiuwu Chen. 2024a. Autocoder: Enhancing code large language model with aiev-instruct. *ArXiv*, abs/2405.14906.

Chao Lei, Yanchuan Chang, Nir Lipovetzky, and Krista A. Ehinger. 2024b. Planning-driven programming: A large language model programming workflow. *ArXiv*, abs/2411.14503.

G. Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for "mind" exploration of large language model society.

Feng Lin, Dong Jae Kim, and Tse-Husn Chen. 2024. Soen-101: Code generation by emulating software process models using large language model agents.

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.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large language model-based agents for software engineering: A survey. *ArXiv*, abs/2409.02977.

Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2024. Agentfl: Scaling llm-based fault localization to project-level context. *ArXiv*, abs/2403.16362.

Zeeshan Rasheed, Malik Abdul Sami, Kai-Kristian Kemell, Muhammad Waseem, Mika Saari, Kari Systa, and Pekka Abrahamsson. 2024a. Codepori: Large-scale system for autonomous software development using multi-agent technology. *ArXiv*.

Zeeshan Rasheed, Malik Abdul Sami, Muhammad Waseem, Kai-Kristian Kemell, Xiaofeng Wang, Anh Nguyen, Kari Systä, and Pekka Abrahamsson. 2024b. Ai-powered code review with llms: Early results. *ArXiv*, abs/2404.18496.

Yashar Talebirad and Amirhossein Nadiri. 2023. Multi-agent collaboration: Harnessing the power of intelligent llm agents. *ArXiv*, abs/2306.03314.

Wei Tao, Yucheng Zhou, Wenqiang Zhang, and Yu-Xi Cheng. 2024. Magis: Llm-based multi-agent framework for github issue resolution. *ArXiv*, abs/2403.17927.

Haoxin Tu, Zhide Zhou, He Jiang, Imam Nur Bani Yusuf, Yuxian Li, and Lingxiao Jiang. 2023. Isolating compiler bugs by generating effective witness programs with large language models. *IEEE Transactions on Software Engineering*, 50:1768–1788.

Lei Wang, Chengbang Ma, Xueyang Feng, Zeyu Zhang, Hao ran Yang, Jingsen Zhang, Zhi-Yang Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji rong Wen. 2023. A survey on large language model based autonomous agents. *ArXiv*, abs/2308.11432.

Yanlin Wang, Wanjun Zhong, Yanxian Huang, Ensheng Shi, Min Yang, Jiachi Chen, Hui Li, Yuchi Ma, Qianxiang Wang, and Zibin Zheng. 2024. Agents in software engineering: Survey, landscape, and vision. *ArXiv*, abs/2409.09030.

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

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Qin Liu, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, and 11 others. 2023. The rise and potential of large language model based agents: A survey. *ArXiv*, abs/2309.07864.

Chun Xia and Lingming Zhang. 2023. Conversational automated program repair. *ArXiv*, abs/2301.13246.

Daoguang Zan, Ailun Yu, Wei Liu, Dong Chen, Bo Shen, Wei Li, Yafen Yao, Yongshun Gong, Xiaolin Chen, Bei Guan, Zhiguang Yang, Yongji Wang, Qianxiang Wang, and Li zhen Cui. 2024. Codes: Natural language to code repository via multi-layer sketch. *ArXiv*, abs/2403.16443.

Simiao Zhang, Jiaping Wang, Guoliang Dong, Jun Sun, Yueling Zhang, and Geguang Pu. 2024a. Experimenting a new programming practice with llms. *ArXiv*, abs/2401.01062.

Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. 2024b. A survey on the memory mechanism of large language model based agents. *ArXiv*, abs/2404.13501.

Cogito, ergo sum: A Neurobiologically-Inspired Cognition-Memory-Growth System for Code Generation

Appendix

A Agent Prompt Details

Here are the prompts for different roles. For certain datasets, there are some changes in their data format.

Prompt for HumanEval, MBPP

Planning phase prompt template:

```
prompt = (
          f"Provide guided steps to solve the following problem
             and identify potential challenges.: {question}. "
          f"[requirement]: less text, don't give code"
)
```

Coding phase prompt template:

```
prompt = (
          f"As a code expert, according to the guidance:{
             design_solution}"
          f"please provide a python solution to the following
             programming problem: {question}."
          f"Ensure that the answer produced by your code matches
             the test cases in the examples:{test_case}"
          f"[Important]only give the code and should not include
             any explanations or comments. "
)
```

Debugging phase prompt template:

```
prompt = (
          f"According to the {question}, the code given is:{
             implementation_solution} "
          f":Fix it using traceback:{result_traceback}. "
          f"[Important]Only give code don't analyze and no
             annotation"
)
```

Super-Role's prompt template:

```
prompt = (
          f"According to the problem:{question}"
          f"Use the experience to give the code to solve it, make
              sure it will pass the text case:{test_case}"

)
```

Super-Role's refinement template:

```
prompt = (
          f"For this problem, {question}, your previous answer
             encountered an error: {first_solution}. "
          f"Traceback: {result}. "
          f"To proceed, ensure the new solution meets the
             following requirements:\n"
          f"1. Is fundamentally different from the previous
             solution.\n"
          f"2. Fixes the above error.\n"
          f"3. Passes all the given test cases: {test_case}.\n\n"
          f"Here are some examples: {Example}. "
          f"Hint: Try to explore different logic or structures,
             such as using loops, functions, or list
             comprehensions.\n\n"
)
```

Prompt for APPS

Planning phase prompt template:

```
prompt = (
          f"Provide guided steps to solve the following problem
             and identify potential challenges.: {question}. "
          f"[requirement]: less text, don't give code"
)
```

Coding phase prompt template:

```
prompt = (
          f"As a code expert, according to the guidance:{
             design_solution}"
          f"please provide a python solution to the following
             programming problem: {question}."
          f"Ensure that the answer produced by your code matches
             the test cases in the examples:{test_case}"
          f"The function name must be the same as in the problem{
             prompt_name}"
          f"[Important]only give the code and should not include
             any explanations or comments. "
)
```

Debugging phase prompt template:

```
prompt = (
          f"According to the {question}, the code given is:{
             implementation_solution} "
          f":Fix it using traceback:{result_traceback}. "
          f"[Important]Only give code don't analyze and no
             annotation"
          f"Make sure the function name is the same as in the
             problem{prompt_name}"

)
```

Super-Role's prompt template:

```
prompt = (
          f"According to the problem:{question}"
          f"Use the experience to give the code to solve it, make
              sure it will pass the text case:{test_case}"
          # f"Use the same function name in the problem{
             prompt_name}"
          f"[Important]:Only codes. No comments or annotation"
)
```

Super-Role's refinement template:

```
prompt = (
          f"For this problem, {question}, your previous answer
             encountered an error: {first_solution}. "
          f"Traceback: {result}. "
          f"To proceed, ensure the new solution meets the
             following requirements:\n"
          f"1. Is fundamentally different from the previous
             solution.\n"
          f"2. Fixes the above error.\n"
          f"3. Passes all the given test cases: {test_case}.\n\n"
          f"Here are some examples: {output}. "
          f"Hint: Try to explore different logic or structures,
             such as using loops, functions, or list
             comprehensions.\n\n"
          f"[requirement]: Only codes. No comments or annotation"
          f"Use the same function name in the problem{prompt_name
             }"
)
```

Prompt for xCodeEval, CodeContest

Due to our use of a unified test, we have forced both the input and output to be a single string parameter. While this approach standardizes the operation, it does not guarantee 100% success in defining the function, which can lead to discrepancies between the test results and reality. We recommend integrating a standard test and removing the forced content to achieve better results.

Planning phase prompt template:

```
prompt = (
        f"Provide guided steps to solve the following problem
            and identify potential challenges.: {question}. "
        f"[requirement]: less text, don't give code"
)
```

Coding phase prompt template:

```
prompt = (
        f"As a code expert, according to the guidance:{
            design_solution}"
        f"please provide a python solution to the following
            programming problem: {question}."
        f"Ensure that the answer produced by your code matches
            the test cases in the examples:{test_case}"
        f"[Important]only give the code and should not include
            any explanations or comments. "
        f"[Important]:Use a function to solve the problem,
            ending with a return.All the code is inside the
            function."
        f"Make sure the function only requires a single string
            parameter."
)
```

Debugging phase prompt template:

```
prompt = (
        f"According to the {question}, the code given is:{
            implementation_solution} "
        f":Fix it using traceback:{result_traceback}."
        f"[Important]Only give code don't analyze and no
            annotation"
        f"[Important]:Use a function to solve the problem,
            ending with a return."
        f"Make sure the function only requires a single string
            parameter.All the code is inside the function."
        f"Only code no comments or other things"
)
```

Super-Role's prompt template:

```
prompt = (
        f"According to the problem:{question}"
        f"Use the experience to give the code to solve it, make
             sure it will pass the text case:{test_case}"
        f"[Important]:Only codes. No comments or annotation"
        f"Use a function to solve the problem, ending with a
            return, and only require a single string parameter"
        f"All the code is inside the function."
)
```

Super-Role's refinement template:

```
prompt = (
    f"For this problem, {question}, your previous answer
       encountered an error: {first_solution}. "
    f"Traceback: {result}. "
    f"To proceed, ensure the new solution meets the
       following requirements:\n"
    f"1. Is fundamentally different from the previous
       solution.\n"
    f"2. Fixes the above error.\n"
    f"3. Passes all the given test cases: {test_case}.\n\n"
    f"Here are some examples: {output}. "
    f"Hint: Try to explore different logic or structures,
       such as using loops, functions, or list
       comprehensions.\n\n"
    f"[requirement]: Only codes. Make only require a single
        string parameter"
    f"All the code is inside the function."
    f"code only require a single string parameter"  #
       codecontest
)
```

B Supplementary Experimental Details

B.1 Datasets

For convenience, we used the HumanEval dataset from Mapcoder (Islam et al., 2024), which contains a sample column that separately extracts the execution examples provided in the prompt, making it easier to execute and return results. Similarly, in MBPP, they also select some data from the test set as inputs, but maintain the independence of the test set and the exclusivity between MBPP and MBPP-ET. For CodeContest, we only use the test section consisting of 165 problems. APPS and xCodeEval utilize a subset of problems extracted from the raw data by MapCoder.

B.2 Evaluation Metric

The unbiased version of **Pass@k** is a widely adopted metric for evaluating the effectiveness of recommendation or code generation systems, particularly under biased sampling or uneven relevance distributions. It estimates the probability that at least one correct item appears among the top-k results, while adjusting for inherent biases such as popularity skew.

Formally, the metric is defined as:

$$\text{Pass@}k = \mathbb{E}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right], \quad (2)$$

where n is the number of generated candidates, c is the number of correct candidates, and k is the number of top predictions considered. The term $\frac{\binom{n-c}{k}}{\binom{n}{k}}$ computes the probability that none of the top-k outputs are correct; subtracting this from 1 yields the probability that at least one correct solution is included.

By averaging over a set of problems, the expectation $\mathbb{E}_{\text{Problems}}$ ensures robustness and generalization across diverse test scenarios. This approach mitigates the tendency of traditional metrics to overestimate performance due to frequent recommendation of a small number of highly probable items.

Pass@1 for One-Shot Evaluation. In our experiments, we adopt **pass@1**, which measures the probability that the model generates a correct solution in a single attempt—a critical capability in real-time or resource-constrained settings.

Let $D = \{x_1, x_2, \ldots, x_N\}$ denote a dataset of N programming problems. For each $x_i \in D$, the model outputs one candidate solution $\hat{y}_i$, which is then executed against a predefined test suite. Define an indicator function $\mathbb{I}[\hat{y}_i \text{ is correct}]$, which equals 1 if the solution passes all test cases and 0 otherwise. The *pass@1* score is then:

$$\text{pass@}1 = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}[\hat{y}_i \text{ is correct}] \quad (3)$$

In settings allowing multiple generations per problem, and assuming c out of n are correct, the expected *pass@k* is approximated by:

$$\text{pass@}k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \quad (4)$$

In particular, for $k = 1$, this reduces to:

$$\text{pass@}1 = \frac{c}{n} \quad (5)$$

This formulation provides a fairer and more robust evaluation of a model's ability to generate correct outputs across a variety of tasks.

B.3 Baselines

We evaluate our approach by comparing it with several baseline methods. First, we use the **Direct** Method, where the prompt is submitted to the LLM without decomposition to assess its intrinsic reasoning. We then evaluate two structured reasoning methods: Chain-of-Thought (**CoT**), which solves the problem step-by-step, and **Self-Planning**, which separates planning and implementation phases. Our approach, which incorporates GitHub searches for relevant code, is compared with **Analogical Reasoning**, a retrieval-based method. **Mapcoder**, a former state-of-the-art method, as a benchmark. Finally we include **CodeSim** a multi-agent framework that enhances code generation through step-by-step input/output simulation. All tests are conducted using GPT-3.5-turbo (GPT-3.5-turbo-0125) and GPT-4 (GPT-4-0613) from OpenAI.

B.4 An example answer for 5 methods of HumanEval #92

We present a detailed comparative analysis of solutions generated by various methods for the 92nd problem in the HumanEval benchmark (Figure 7). Among all evaluated approaches, only **Cogito** successfully passes the initial test case. The test, designed to verify functional correctness, checks whether the output of the candidate function satisfies the condition `candidate('TEST')`

== 'tgst'; all baseline models fail this basic requirement.

For each method, we highlight the specific errors in their generated outputs and provide concise explanations of the underlying failure modes. These typically include incorrect string manipulations, misunderstanding of character transformations, or misinterpretation of input-output constraints. Such errors underscore the challenges existing methods face when dealing with nuanced program logic or subtle pattern recognition.

In contrast, Cogito not only produces a correct solution but also demonstrates consistent reasoning throughout its multi-stage workflow. We present its outputs across different phases of generation, illustrating how it iteratively refines its understanding and progressively improves the solution. This example showcases Cogito's capacity to coordinate planning, coding, and self-correction, enabling it to outperform traditional single-pass generation methods.

B.5 The comparison between GPT-3.5-TURBO and GPT-4 responses.

We analyze model performance on the HumanEval and APPS benchmarks by comparing responses from GPT-3.5 and GPT-4. To better understand the limitations of earlier models and the improvements in newer ones, we specifically focus on instances where GPT-3.5 fails while GPT-4 produces correct solutions. This targeted comparison allows us to examine the underlying causes of failure and highlight the differences in reasoning and generation strategies between the two models.

We begin with two examples from the **HumanEval** benchmark—Problems #32(see Figure 8) and #92(see Figure 9). In both cases, GPT-3.5 explores multiple strategies but ultimately fails to solve the tasks correctly. In contrast, GPT-4, building upon prior trial-and-error attempts, is able to arrive at the correct solution. These examples illustrate GPT-4's improved capability to incorporate feedback and refine its reasoning over successive generations.

The next two examples are drawn from problems #1628 and #3531 in the **APPS** dataset, which contains a broader and more diverse set of programming tasks, including algorithmic and implementation-focused challenges (see Figure 10, Figure 11). Similar to the HumanEval case, GPT-3.5 struggles to provide a correct answer, often producing incomplete or logically flawed code. In both cases, the Super-Role, during the planning phase, successfully leveraged prior learning to provide guidance that explicitly avoided common pitfalls. As a result, the final solutions were correct, demonstrating the effectiveness of accumulated experience in enhancing decision-making and task performance.

These examples collectively highlight the advancements of GPT-4 in terms of *code generation accuracy, problem decomposition, and syntactic correctness*. The improvements are especially evident on tasks requiring multiple reasoning steps or understanding of non-trivial control flow, where GPT-3.5 tends to underperform. While anecdotal in nature, these qualitative cases complement our quantitative results and provide concrete insights into where the improvements of GPT-4 manifest in practice.

B.6 The complete response process of Cogito in APPS #1628

Case Study: Task 1628 in APPS. We present a comprehensive case study of Task 1628 from the APPS benchmark, illustrating the complete reasoning and generation process undertaken by **Cogito**. Figure **??** displays the full sequence of responses, including those from all roles involved in the multi-agent workflow.

To contextualize the development of the final solution, we also include the intermediate responses exchanged between adjacent roles within each group. These interactions provide insight into the causal relationship between design decisions, implementation strategies, and debugging feedback, thereby elucidating the contribution of each stage to the overall outcome.

Furthermore, we annotate each response with corresponding errors, modifications, and correct segments. For every modification, we detail the underlying rationale, explaining whether the change was necessary for correctness, performance, or clarity. This fine-grained analysis allows us to track the evolution of the code and pinpoint the precise factors that led to success or failure in each iteration.

By dissecting these role-based interactions and their iterative refinements, this case study offers a transparent view into the collaborative dynamics of Cogito's problem-solving process and provides qualitative evidence of its multi-role reasoning capabilities.

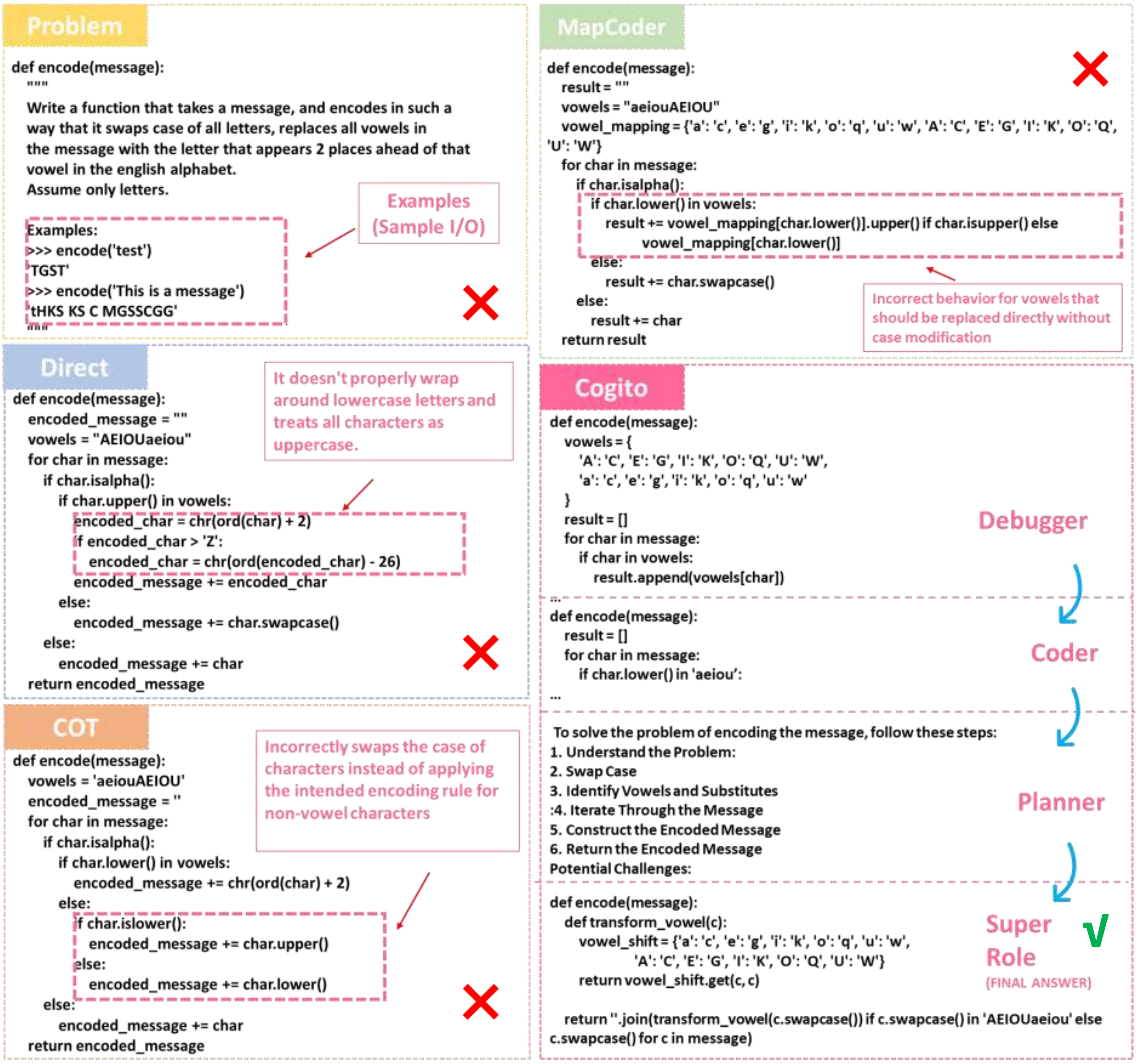


Figure 7: An example answer for 5 methods of HumanEval #92.

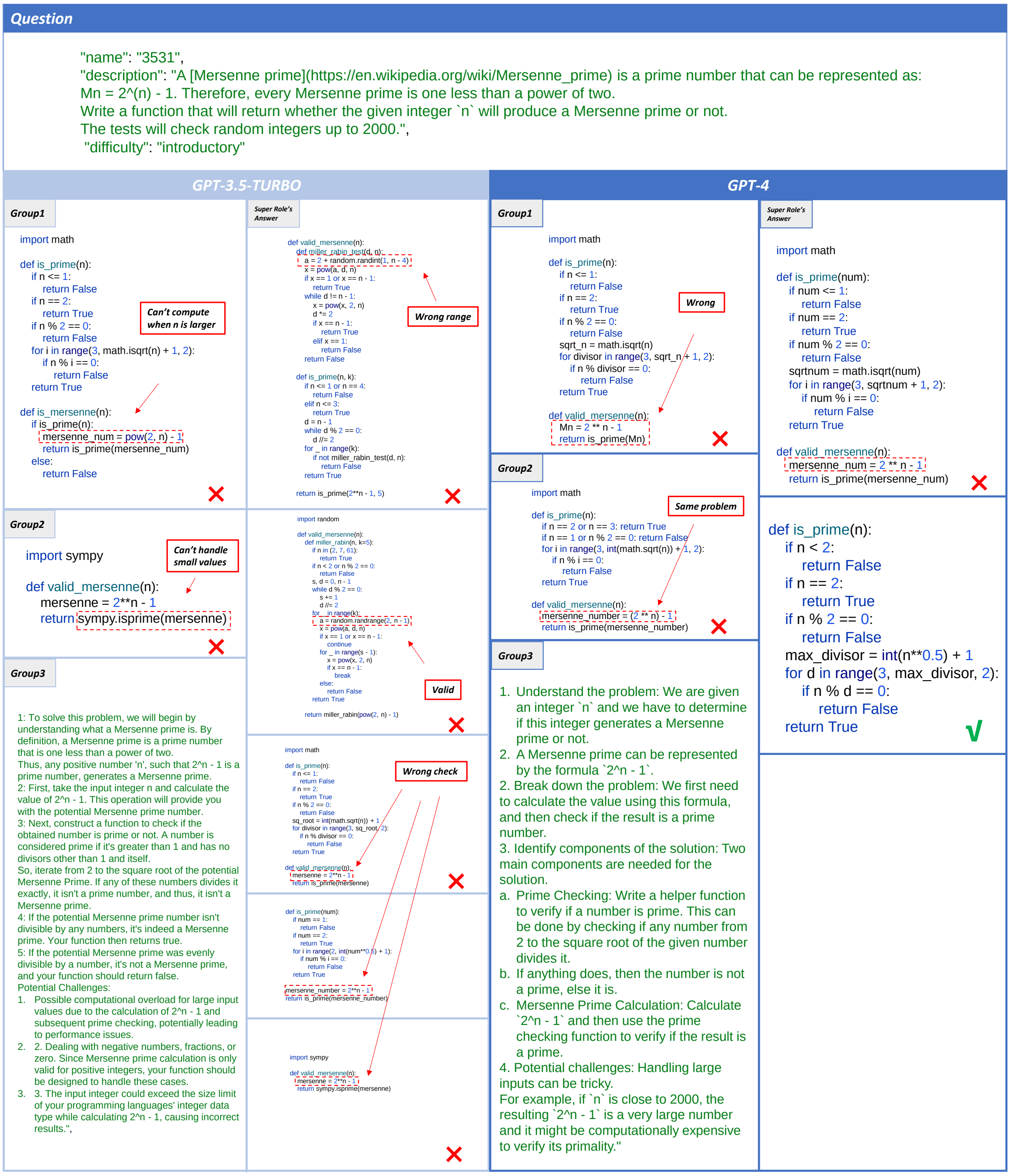


Figure 8: The comparison of answers between GPT-3.5-TURBO and GPT-4 in HumanEval#32.

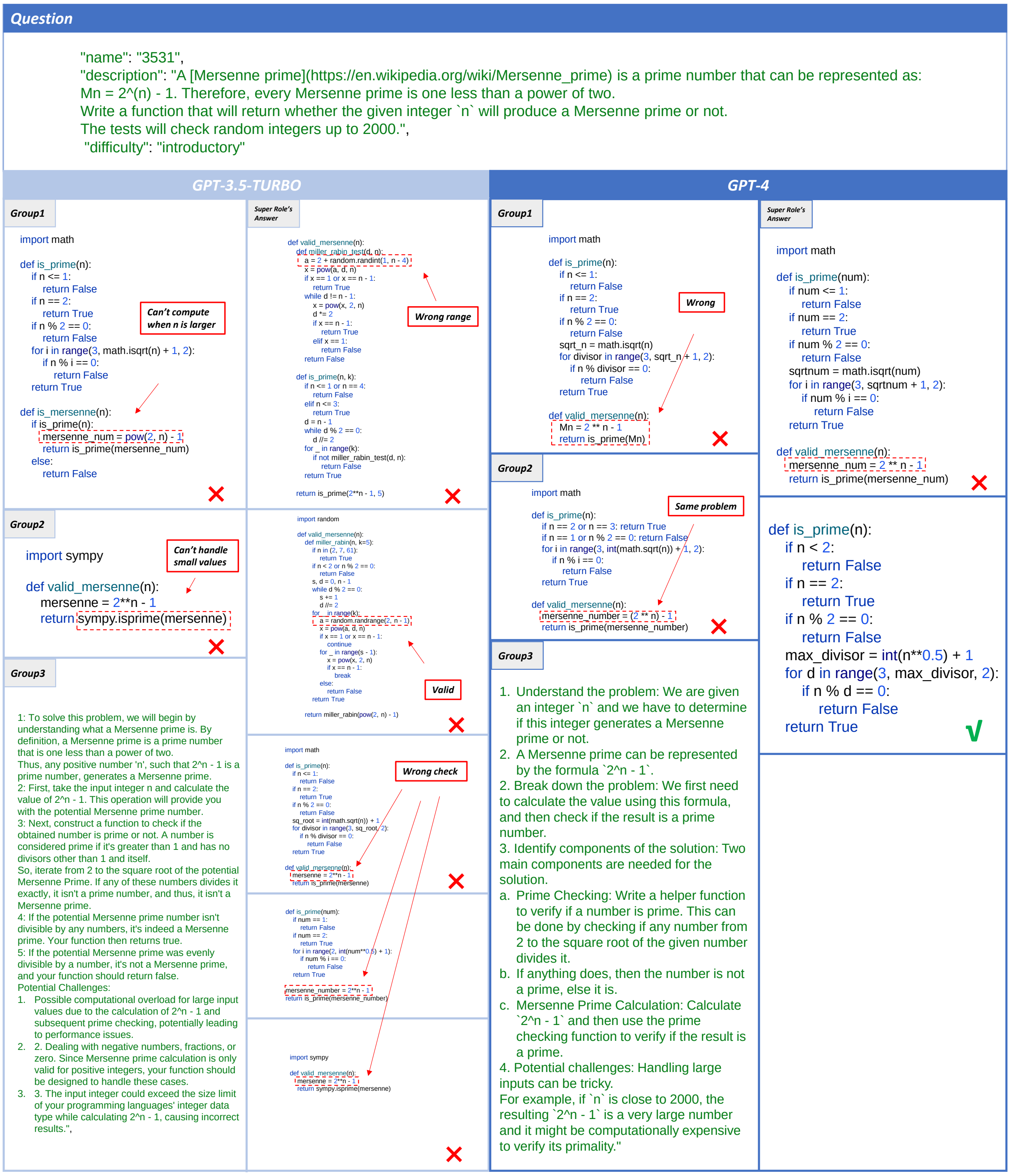


Figure 9: The comparison of answers between GPT-3.5-TURBO and GPT-4 in HumanEval#92.

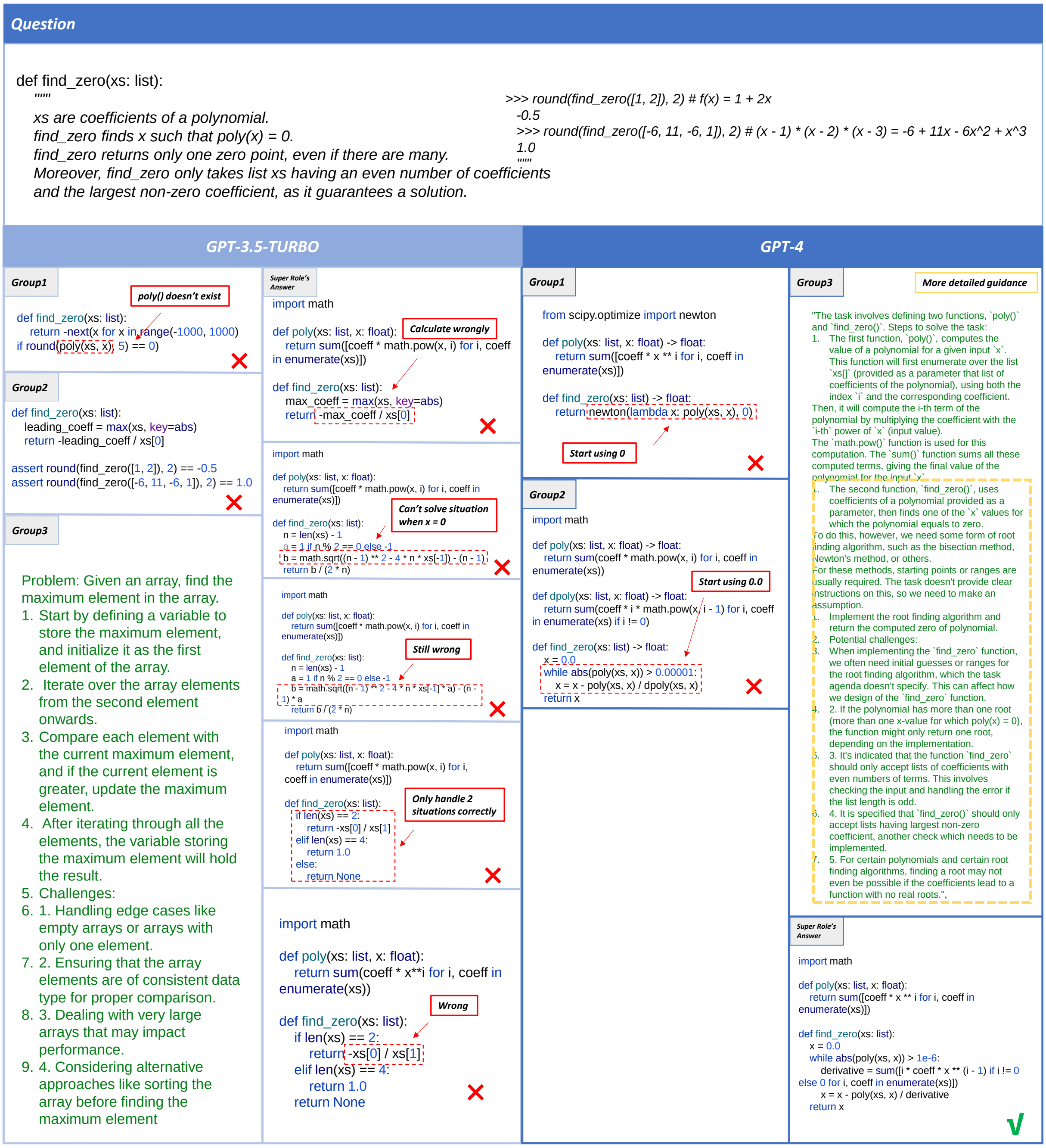


Figure 10: The comparison of answers between GPT-3.5-TURBO and GPT-4 in in APPS#1628.

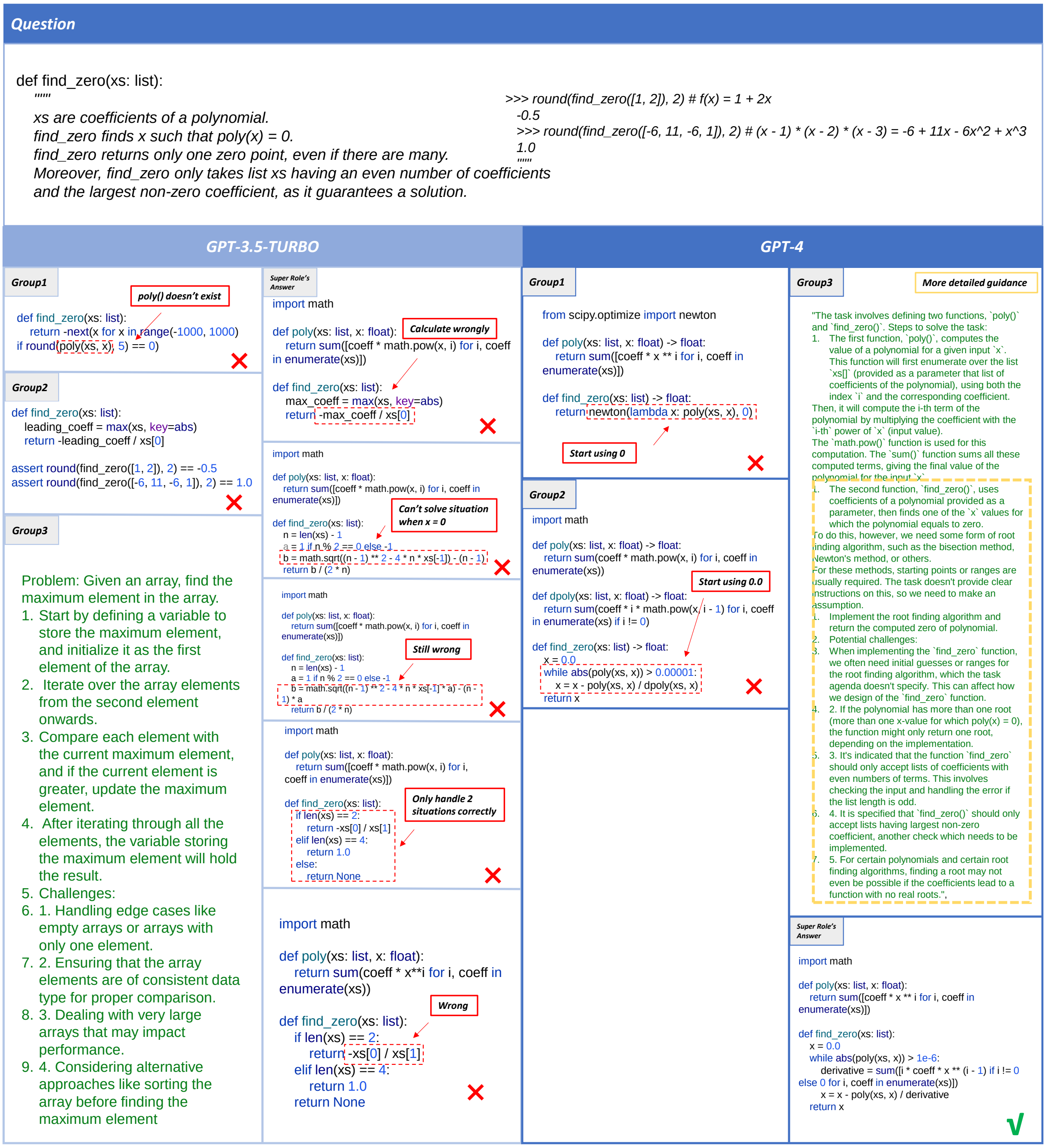


Figure 11: The comparison of answers between GPT-3.5-TURBO and GPT-4 in APPS#3531

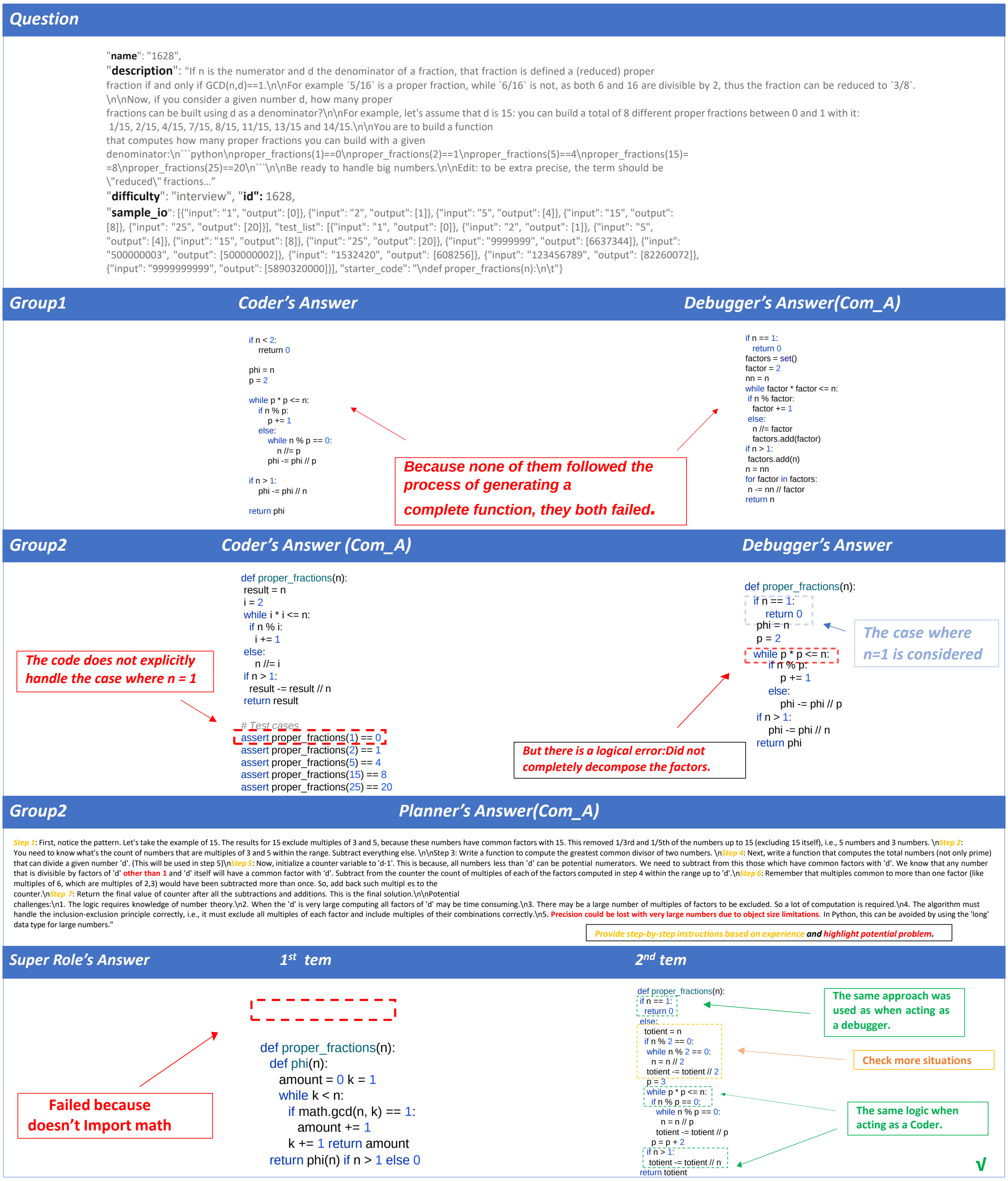


Figure 12: The complete solution process of Cogito in APPS#1628.