BlockChain Coder: The Multi-Agent Framework Intellectualize LLM on Code Generation

Anonymous TACL submission

Abstract

Large Language Models (LLMs) have demonstrated surprising performance in solving complex coding problems which require substantial reasoning, planning, and refining abilities. However, potential data contamination aroused the hesitation about whether LLMs barely memorized the answer from its training data. To this end, we conduct an experiment comparing the performance of LLMs on LeetCode problems released long ago versus those released more recently. We observed on average over 40% accuracy drop which echoes the hesitation. To guide the LLM in adapting to unseen problems, we propose the BlockChain Coder, a multi-agent framework aiming at solving coding problems in two approaches. The multi-agent framework consists of a Planner, an Optimizer, and a Debugger. There are two approaches to generating sequential code blocks, namely a unit test approach and a question-based approach. The unit test approach uses public test cases as the ground truth and accordingly generates code blocks. While the question-based approach generates high-quality step-by-step code solutions through structurally retrieving and refining the problem information. The Blockchain Coder achieved significant improvement in accuracy when tackling both the latest and typical LeetCode problems, indicating the multi-agent framework empowered the LLM to generalize its inherent knowledge to solve unseen complex problems.

1 Introduction

According to classical definitions from the psychology domain, the reasoning, planning, and refining abilities of autonomous agents are indicators of general intelligence. (Gottfredson, 1997; Wechsler, 1941; Humphreys, 1979). Combining such abilities, an intelligent agent expresses so-called Goal-directed adaptive behavior (Sternberg, 1982; Feurerstein et al., 2002) meaning to adapt past knowledge and experience into unseen scenarios. Recently, the tremendous success of Large Language Models (Guo et al., 2024; Ye et al., 2023; Achiam et al., 2023) on classical coding benchmarks (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021; Yin et al., 2018) seems evident that current LLMs are intelligent enough to solve complex problems as those coding problems require substantial reasoning, planning and refining abilities of human programmers to solve. Some (Huang et al., 2023c) argue that competitive-level coding problems such as problems appearing in ICPC contests can be used to evaluate LLMs' abilities to solve complex problems. However, a recent study (Riddell et al., 2024) on the quantization of LLMs' data contamination revealed that nearly 20% of problems in the MBPP and HumanEval benchmarks have been leaked in the training phase of a subset of LLMs. Further studies (Liu et al., 2023; Tian et al., 2023) echo this point by that researchers utilized the GPT series to do Leetcode problems, and they found that there is a cliff accuracy drop ($70\% - 80\%$ drop) between Leetcode problems released before 2022 and after 2022. These studies aroused wide hesitations about whether the LLM's surprising coding performance comes from memorizing its inherent knowledge rather than intelligently solving complex problems. To this end, we propose two research questions to investigate.

RQ 1. Is it true that LLMs perform significantly worse for coding problems released after their training phase?

RQ 2. How can we guide LLMs to perform better

on unseen coding interview problems? Such that LLMs can generalize the inherent knowledge to unseen problems. It would be a step toward true intelligence.

To address ***RQ 1***, we randomly selected 100 older LeetCode problems released before 2022, along with the most recent 100 new LeetCode problems as a new coding benchmark. Although prior studies (Liu et al., 2023; Tian et al., 2023) have collected a similar dataset, their dataset size is small. We conducted an empirical experiment to test the accuracy of popular LLMs on both older and new problems. The results show, on average, a more than 40% accuracy drop in new LeetCode problems across all levels of problem difficulty and all selected LLMs. The finding in Table 1 echoes the memorization theory.

	Deepseek	GPT-3.5	GPT-4
Old Easy	100.0	93.33	100.0
Old Medium	72.5	65.0	80.0
Old Hard	53.33	53.33	83.33
New Easy	56.67	56.67	66.67
New Medium	5.0	5.0	7.5
New Hard	3.45	0.0	3.45

Table 1: Performance drop on problems across all difficulty levels

To address ***RQ 2***, we propose the BlockChain Coder, a multi-agent framework with two powerful approaches to intelligently guide LLMs on code generation task. Starting from Chain-of-thought prompt (Wei et al., 2022), many search-based prompt frameworks (Yao et al., 2024; Hao et al., 2023; Yao et al., 2022; Radhakrishnan et al., 2023) have boosted LLMs on reasoning, planning and acting. The core underlying in-sight is to decompose a complex coding problem into simpler steps by logical step-by-step plan and then refine the plan to generate a satisfactory solution. Similar idea is applicable to the field of code generation. Specifically, the BlockChain Coder framework views the coding problem in two approaches, the unit-test based approach and the question-based approach. Each approach has its own Planner agent and Optimizer agent. The Unit-test based approach relies on a detailed comprehension of the public test cases to plan the code generation and code optimisation. The question-based approach guides the code generation through iterative questioning on structured information flow. Then, both approaches use the same code-execution based Debugger agent to achieve robust debugging. Each agent is designed with unique functionality to gradually solve a complicated coding problem from scratch.

Our experiment results show that the quality of generated code solution from our framework outperforms a lot of single Large Language Model.To summarize our contribution:

1. We conducted an empirical experiment to test the multi LLM-agent framework ability on solving unseen coding problems from the LeetCode website.

2. We proposed a robust multi-agent framework in two effective approaches to enhance Large Language Models' capabilities on code generation task. The BlockChain Coder effectively and innovatively use multiple-step strategy to separate coding questions into simpler steps and solve them step-by-step with logical reasoning and justification. The overall performance of BlockChain surpasses baseline LLM by a large margin.

3. The Blockchain Coder explores two different approaches which views the given problem different to previous works. It opens a gate for future research of our proposed framework in other tasks.

2 Related Work

2.1 Language Model as Code Generator

The Large Language Models are trained with both textual corpus and code corpus. They have natural coding abilities. Many prior works explored augmenting the coding ability of LLMs through testing, searching, planning, or modularization. AlphaCode (Li et al., 2022) and MBR-Exec (Shi et al., 2022) proposed to use sophisticated rule-based methods to rank a set of auto-generated codes through execution tracing. Similarly, Alphacodium (Ridnik et al., 2024) explicitly asks LLM to self-rank a list of coding options, and generates additional test cases to guide the code iteration. CodeT (Chen et al., 2022) specifically pre-trained a language model to both generate solutions and test suites. However, existing methods of

using general-purpose LLMs to choose programming options and generate extra test cases still suffer from severe hallucinations.

Given a natural language coding requirement, Self-planning (Jiang et al., 2023) uses a planner to devise a step-by-step plan to outline the following code implementation. MoTCoder (Li et al., 2023) and CodeChain (Le et al., 2023) decompose coding tasks into logically independent sub-modules in a non-monolithic manner. Compared to prior works which directly feed the problem description to LLMs to generate a solution, we adopt a similar planning method to modularize the task into coding blocks, but explicitly mining implicit knowledge underlying the given problem and public test cases.

2.2 Language Model as Debugger

Recently, more related works have been proposed to boost generation quality through iterative self-revisions. Self-correct (Welleck et al., 2022) and CodeRL (Le et al., 2022) introduce secondary models to predict the correctness of output programs and revise them accordingly. Self-refine (Madaan et al., 2024), and Reflexion (Shinn et al., 2024) propose to facilitate better code revision with synthetic natural language explanation or reflection self-generated by LLMs. Self-repair (Olausson et al., 2023) and ILF (Chen et al., 2023) follow a similar strategy but highlight the use of natural language explanation provided by human experts. Different from prior approaches, we propose to generate more modularized programs, monitoring the run-time executions and variable updates to reason the error.

2.3 Multi-agent Code Generation

Autonomous agents can make decisions, use tools, and automatic reasoning. Given the rise of LLMs in recent years, many studies attempted to build autonomous agents for the coding task, but none of them are designed for interview-level coding problems. MetaGPT (Hong et al., 2023) is a representative of such studies, it follows the Standardized Operating Procedures (SOPs) of human experts to build six agents that can accomplish the software development process in an end-to-end manner. AgentCoder (Huang et al., 2023a) uses a programmer agent, a test designer agent, and a test executor agent to respond to the verbal programming requirement via iterative testing and optimisation. We are the first to work to empower LLM to solve complex interview problems in a multi-agent setting.

3 Methods: BlockChain Coder

3.1 Problem Formulation

We define the code generation task as a conditioned token generation task. Given a sequence D which represents sequential tokens in LeetCode official problem description, and few public test cases

$$T_M = \{(i_m, o_m, e_m)\}_{m=1}^{M}$$

where M is the total number of public test cases, i_m, o_m, e_m are input tokens of LLMs, expected generated output tokens of LLMs and official generated test case explanation tokens from LLMs respectively. The Language Model receives D and generate sequential solution tokens

$$\hat{S} = (\hat{s}_1, \hat{s}_2, ...\hat{s}_T), \hat{s}_{t=1,...,T} \in P$$

where P is a collection of python code tokens. Through the auto-regressive nature of LLMs, the final solution $\hat{S}$ is composed by iterative sampling single code token $\hat{s}_t$ using

$$P_\theta(\cdot|\hat{s}_{1:t-1}, T_M, D)$$

A solution $\hat{S}$ is accepted by LeetCode only if it can pass all official private test cases $(i_j, o_j)_{j=1}^{J}$ by

$$\hat{S}(i_j) = o_j, j \in 1, .., J$$

To solve the problem, we define three autonomous agents: A_P planner agent, A_O optimizer agent and debugger agent A_D.

In our multi-agent framework, we introduce the two distinct but both effective approaches below to solve the coding question. The reason for having these two different approaches is that our framework aims to solve coding questions from the LeetCode website where the LeetCode website provides two parts of information as our input: the coding questions and their associated test cases. Hence, we want to explore how to utilize the question and its associated test cases to guide LLMs to solve coding questions. The Unit-Test Based Approach focuses on utilizing the test cases to enhance LLMs' reasoning capability and the Question-Based Approach pays more attention to extracting essential information directly from the question description to construct a logical step-by-step plan to guide the generation of correct code solutions in our proposed framework.

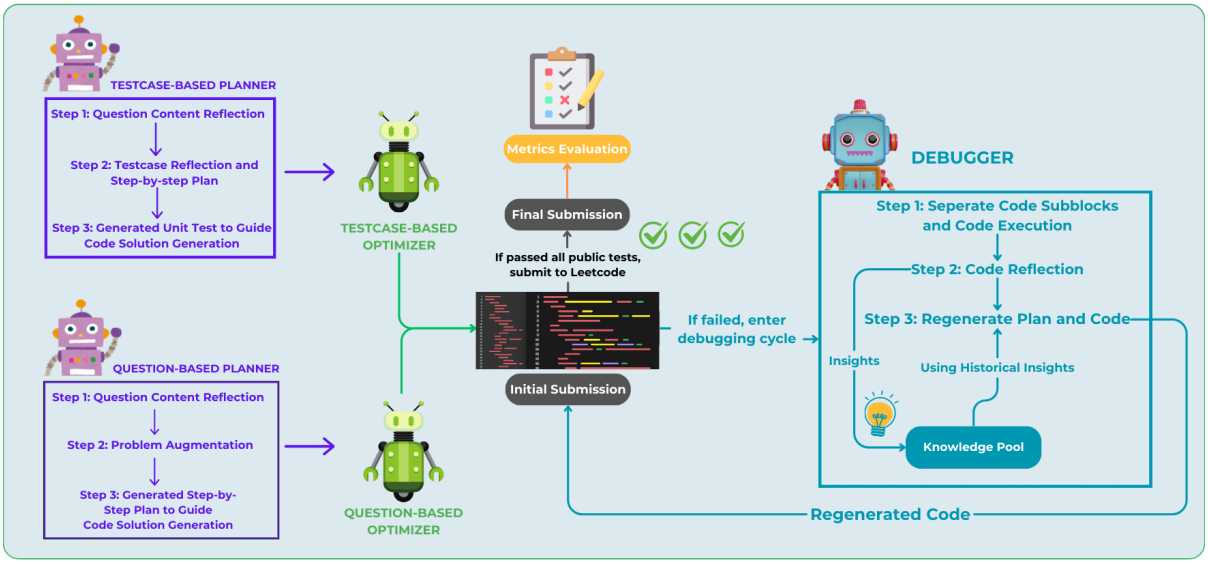


Figure 1: A graphical demonstration of the proposed Multi-Agent coding framework where the coding problems will be processed by **Test-based Planner** or **Question-based Planner** to generate a code solution and then their associated **Optimizer Agent** will optimise the code solution and then attempt to locally test the code solution. If the code solution passes all public tests, it will be submitted to Leetcode directly. If the code solution does not pass public tests, the **Debugger Agent** will be activated to iterative refine and fix the code solution with the tool of Knowledge Pool until the code solution passes all public tests or it reaches the maximum debugging times.

3.2 Unit-Test Based Approach

3.2.1 BlockChain Planner

Upon receiving two token sequences D and T_M. The planner first makes a self-reflection to clearly explain the involved background concepts, and tackle potential ambiguities in the problem description through verbal comparison with test case explanations. Such self-reflection yields an augmented problem description $\mathrm{aug}(D)$ with all ambiguity clarified.

Inspired by (Jiang et al., 2023; Li et al., 2023; Le et al., 2023), we devise a detailed step-by-step programming plan directly using all public test case explanations. Similar to prior augmentation, the planner A_P is required to augment the original test case explanations e_m, $m \in \{1, \ldots, M\}$ through detailed self-reflection. Then, the planner will generate the programming plan $plan$ which decomposes the complex problem into easy-to-implement code blocks based on this augmented test case explanation. The generation of planning steps are denoted by

$$\text{plan}_{\,i} \sim P_{A_P}\left(\cdot \mid \text{plan}_{1:i-1}, \text{aug}\left(e_m\right), \text{aug}(D)\right)$$

The plan needs to be filled with actual Python functional codes.

With the step-by-step plan in mind, as some test cases provide the calculation process about how the input leads to the expected output in very detail, we adopt the calculation as a ground truth to generate both actual functional codes and unit tests for each function. In this approach, we largely reduced the hallucination that frequently occurred in prior studies using AI-generated test cases, and also reinforced the generated function to follow the intention of the programming plan. These generated Python tokens together with unit tests are the output of the planner agent.

$$\hat{s}_t, u_t \sim P_{A_P}(\cdot \mid \hat{s}_{1:i-1}, plan, \mathrm{aug}(e_m))$$

Noticeably, we run the above-conditioned generation process for each $T_m = \{(i_m, o_m, e_m)\}, m \in \{1, \ldots, M\}$ independently, and choose the solution that maximizes the accuracy on local tests $\max_{\hat{S}}(\mathrm{Acc}(\hat{S}, T_M))$ for the next steps.

3.2.2 BlockChain Optimizer

The design of the optimizer is intuitive, to further optimize the data representation and the execution time, we use a BlockChain optimizer A_O to iteratively optimize each code block. For each specific function F, we explicitly require the agent to continually sample different code tokens $s'_t \in F'$ that

could possibly accelerate the execution as a candidate substitution F'. Considering the hallucination across all LLMs, the substitution is accepted only if it passes the corresponding unit test u_t. The iterative sampling process stops either the LLM decides no function could be further optimized or the maximum number of iterations is reached.

3.3 Question-Based Approach

(Huang et al., 2023b) discusses the Large Language Models' hallucination, possible hallucination underlying causes, and several approaches to mitigating different kinds of hallucinations. Inspired by this, we propose the question-based coding approach.

3.3.1 BlockChain Planner

The BlockChain Planner is powered by LLMs. After receiving the coding question token sequence D and T_M, the BlockChain Planner will first take a step of self-reflection to give a detailed examination of the given question and test case and subsequently produce useful hints and insights based on the question description and test cases. Based on generated hints H and insights I, the BlockChain Planner will start constructing a step-by-step plan.

$$\text{plan}_{\,i} \sim P_{A_P}\left(\cdot \mid \text{plan}_{1:i-1}, \text{I}, \text{H}\right)$$

Finally, the Planner will generate the code solution for the coding question based on the step-by-step plan from its last step.

$$\hat{s}_t \sim P_{A_P}(\cdot \mid \hat{s}_{1:i-1}, plan)$$

The BlockChain Planner generates a high-quality step-by-step plan in three steps: Step 1: Content Understanding and Refinement, Step 2: Step-by-Step Plan Construction, Step 3: Code Solution Generation. When a coding task is given to the Planner, the Planner first analyzes the coding question D to extract a hint-and-insight token sequence H, where the planner first generates a keyword token sequence K, and forms a detailed problem statement guided by K as a conclusion to the problem background. Then Planner will construct a step-by-step plan based on the augmented problem statement, and subsequently produce a code solution based on the plan.

By using the keyword token sequence K, the Planner effectively reduces the potential concept hallucination as the keywords enforce the agent to generate reliable output in a conceptually correct manner. Following the carefully designed prompts, the Planner will strictly follow correct concepts to construct a step-by-step plan and make the code solution based on previous logical analysis without encountering the issue of overthinking.

As a result, the Planner can produce high-quality code solutions for the coding problem in a one-shot setting.

3.3.2 BlockChain Optimizer

In our experiments, we find that a complicated code solution is more likely to fail to pass the public test cases because the agent is likely to make mistakes and be trapped in reasoning hallucinations. The Optimizer's task is to simplify and optimise the provided code solution from the Planner to improve the overall code quality. The Optimizer takes the original question description token sequence D, public test case token sequences T_M, a solution template T, and the code solution token sequence $\hat{S}$ passed by the Planner as inputs. From its observation of data structure and algorithm selection, the Optimizer will subsequently attempt to make the code solution more concise and efficient. The agent will optimise the code solution only if it finds any data inefficiency or algorithm mismatch from the code solution. In particular, the Optimizer will produce a state token (True/False) at the beginning of each step to decide whether the optimization is required or not. The usage of this state token can effectively instruct the LLM to optimise the code solution by taking a binary decision as the agent can choose not to optimise the code if the agent finds that the code solution is already good enough. The simplification process of the code in the Optimizer focuses on improving the data structure and algorithm selection of the code, it will not lead to information loss like forgotten edge cases. Such an approach guarantees a more efficient code solution by optimizing the time complexity of solutions. Also, it further avoids the LLM's overthinking issue and ensures that it is easy for the latter Debugger to detect possible flaws from a simplified solution.

3.4 BlockChain Debugger

Once the generated solution is still hindered by public tests after the above process, the debugger agent Figure 2 comes into help. As we have a packed solution $\hat{S}$ represented by a chain of different functions, we could trace the run-time exe-

cution by explicitly printing out any variable value updates among different functions. Having the dynamic of an input-output data flow together with the corresponding ground truth of expected output, the debugger can reason the erroneous point R_{error} in the program. Also, we use the Knowledge pool $Knowledge$ as a memory module that accumulates failure debugging experiences in previous debugging rounds, such that the debugger agents are enforced to avoid useless debugging effort. Then, the Debugger correspondingly regenerates the programming plan based on the reasoning. The debugging process is in fact iterative refinements on the plan and solution tokens. The iterative re-sampling tokens on the programming plan and solution tokens will stop either when all public test cases are passed or the maximum number of iterations is reached. Our Debugger exceeds

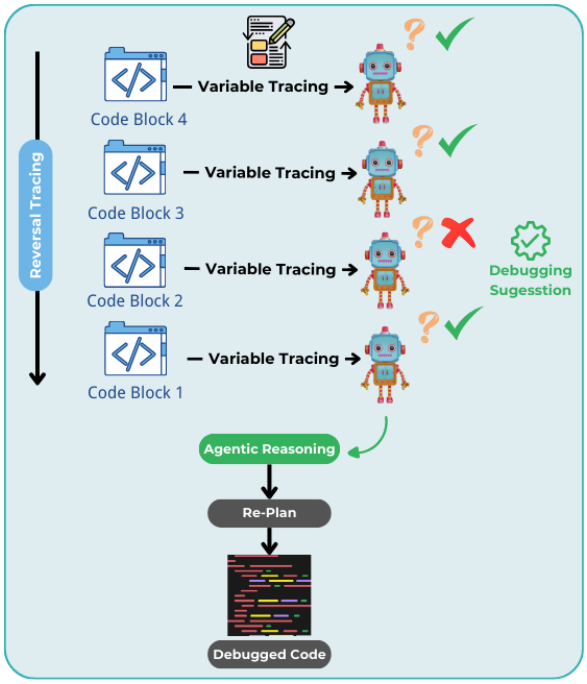


Figure 2: Debugger that traces variable updates in reversal order, the agent checks abnormal data flow from the last code block to the first code block. Empowered by LLMs, the Debugger can reason the cause of errors and re-generate the programming plan as well as the code solution that leads to this error.

previous works on the following aspects: it reason errors through running run-time execution tracing, but does not focus on line-by-line execution like other works, rather tracing the variable updates among code blocks in reversal order. The method largely reduces hallucinations that occur in error tracing. Also, we do not ask the agent to directly fix actual codes, but debugging through regenerating the erroneous programming plan. The method keeps a logical consistency in code generation.

4 Experiment Design

To prevent data contamination on classical coding benchmarks like HumanEval and MBPP, we propose a new benchmark, LeetCodeContest to asses the LLM's actual ability on solving coding problems. The newly collected LeetCodeContest dataset consists of two subsets, one subset is 100 randomly selected LeetCode problems released before 2022, and another subset is 100 randomly selected LeetCode problems released in the past three months. For each subset, there are 30 easy-level problems, 40 medium-level problems, and 30 hard-level problems. Each coding problem consists of a problem description and a few official public test cases. In the development of our multi-agent framework, we selected two API-access LLM models as the foundation model, namely *DeepSeek-Coder* and *gpt-3.5-turbo-1106*.

For the experiment, we compare the single LLM's coding accuracy with the coding accuracy of our multi-agent framework on the LeetCodeContest dataset. We aim to demonstrate the advantages of our approach in solving the challenging leet code problems(with specific strength in unseen medium to hard problems) presented in the dataset.

4.1 Baseline selection criteria

The *DeepSeek-Coder* is a recent LLM that claimed superior coding performance on classical coding benchmarks. It was released for public access on Dec.2023 and provides free usage of ten million tokens for new registers. Similarly, for the *gpt-3.5-turbo-1106*, the model is used as a common baseline in various NLP studies. It was released on Nov.2023. Its costs per token are accepted for us compared to more the sophisticated *GPT-4* series.

4.2 Two Evaluation Metrics

1. **Pass@1 Rate**: Measures the proportion of generated solution that passes the LeetCode official check in one submission. It is a common metric in evaluating code generation

Table 2: Case Pass Rate (CPR) on LeetCodeContests Dataset

Model	Method	Old			New		
		Easy	Medium	Hard	Easy	Medium	Hard
DeepSeek-Coder	Baseline	100.00	76.40	56.47	60.00	16.38	6.70
	Question-Based	100.00	83.40	72.91	80.00	38.71	21.07
	Testcase-Based	98.80	84.30	76.97	80.51	40.24	38.08
ChatGPT-3.5	Baseline	93.33	67.80	56.80	60.00	18.28	34.32
	Question-Based	96.80	78.40	70.91	65.50	25.24	20.83
	Testcase-Based	99.00	78.80	86.25	74.13	34.63	32.80
ChatGPT-4	Baseline	100.00	82.40	84.71	73.00	28.65	26.19

Table 3: Pass@1 Rate on LeetCodeContests Dataset

Model	Method	Old			New		
		Easy	Medium	Hard	Easy	Medium	Hard
DeepSeek-Coder	Baseline	100.00	72.50	53.33	56.67	5.00	3.45
	Question-Based	100.00	82.50	66.67	76.67	25.00	6.90
	Testcase-Based	93.33	80.00	56.67	73.33	22.50	0.00
ChatGPT-3.5	Baseline	93.33	65.00	53.33	56.67	5.00	0.00
	Question-Based	96.67	75.00	70.00	63.33	10.00	0.00
	Testcase-Based	93.33	72.50	60.00	66.67	10.00	0.00
ChatGPT-4	Baseline	100.00	80.00	83.33	66.67	7.50	3.45

abilities.

2. **Case Pass Score (CPS)**: We propose a new metric called the Case Pass Score, which provides a graded measure of success, rather than a binary pass/fail assessment. The CPS is defined mathematically:

$$\text{CPS} = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{r_i}{t_i} \mathbb{I}_i \right)$$

where N is the number of problems in the dataset, r_i is the number of passed private cases for problem i, t_i is the total test cases for problem i, and $\mathbb{I}_i$ is 1 if all public tests pass in problem i, 0 otherwise. For a problem, when all public test cases have passed, the CPS metric calculates the weighted average of the proportion of private test cases that pass. It effectively filters the evaluation to focus only on problems that meet the basic functionality tested by the public cases.

5 Results and Discussion

5.1 Experiment Result

In our experiment, we use the deepseek and GPT 3.5 model as the engine of our multi-agent LLM framework and evaluate the performance of our framework across 200 coding questions. We also conduct an ablation study for our BlockChain Planner and the performance evaluation of four single large language models: Deepseek (DeepSeek-Coder-Base-33B model), GPT 3.5 (gpt-3.5-turbo-1106 model), GPT 4 (gpt-4-1106-preview model).

5.2 Experiment Analysis

5.2.1 Result Analysis

1. Multi-agent Framework using DeepSeek vs GPT models Table 3 shows that our framework, which uses DeepSeek, performs similarly or even better than the Baseline GPT-4 model, particularly with new questions on LeetCode. This demonstrates that our multi-agent LLM framework can effectively narrow the performance gap between different large models, despite DeepSeek having 33 billion parameters compared to GPT-4's more than 1 trillion, even on unseen problems. However, we observed a slight performance drop with Old Easy questions. This decrease could be because our framework encourages reasoning over only memorization, or it might simply be due to randomness in the LLM's performance. Given

more time, we could conduct additional experiments to further investigate these results.

our framework using GPT 3.5, the improvement in the pass@1 rate is not as significant as in CPR (especially in New Hard questions). During experiments, we found that GPT 3.5 had a better interpretation ability of questions than DeepSeek, but it didn't follow the prompts as closely as DeepSeek and had lower baseline accuracy. This discrepancy may explain the lower pass@1 rate.

2. Question-Based vs Testcase-Based
Tables 2 and 3 show that the question-based approach produces a greater improvement in the pass@1 rate, whereas the test-case-based approach improves CPR more effectively. This suggests that the retrieval of a step-by-step plan through the question-based method improves overall understanding of the questions, but has the risk of being too general and not addressing specific problem requirements. In contrast, the test-case-based approach is more closely aligned with the specific test cases, although it may miss certain edge cases not included in public tests, resulting in a lower pass@1 rate.

5.3 Experiment Discussion

5.3.1 Ablation Study

In our experiment, we conduct an ablation study to investigate the contribution of BlockChain Planner to our framework as it directly processes the coding task and delivers the information to the latter agents and the performance of BlockChain Planner hugely affects the overall performance of our multi-agent LLM framework. From our ablation study experiment and Figure 5 and Figure 6 in the Appendix, we find that the BlockChain Planner generally performs better than the baseline LLM's performance in both new and old coding questions by using Case Pass Rate and Pass@1 metrics. Our ablation study proves the positive contribution of our BlockChain Planner.

5.3.2 Ablation Experiment Result Discussion

- The multi-agent LLM framework can effectively enhance the reasoning ability of LLMs on the code generation task. This is because the multi-agent interaction can effectively minimize the potential hallucination from a single LLM and reduce the possible misunderstanding from LLMs.
- The multi-agent LLM framework equipped with different approaches/structures can obtain different benefits. The Question-Based Approach can better extract the essential information from coding questions to increase the successful rate of generating a code solution that passes all test cases. The Unit-Test Based Approach can ensure LLM fits the coding question better by using the unit test on the test cases so the code solution from this approach can pass most of the coding questions' test cases.
- Experiment Limitation: Due to the time constraint and computational resources constraint, we restrict the maximum debugging time to 3 and we do not consider a more sophisticated multi-agent LLM framework. The number of debugging times and the structure of the multi-agent LLM framework can be further investigated in the future.

6 Conclusion

In this project, we design and present BlockChain Code, an innovative multi-agent LLM framework with Unit-Test Based Approach and Question-Based Approach that performs well on the code generation task by using chain of thought prompting techniques and code blocks concept. In the Question Based Approach, we deploy chain-of-thought prompting techniques to convert the coding question into a logical step-by-step plan by gradually extracting and refining the crucial information of the coding question. In the Unit-Test Based Approach, we. Both approaches aim to solve the coding question and minimize the potential LLM hallucinations and they guarantee our multi-agent LLM framework has better performance in different scenarios.

Our experiments indicate an obvious performance improvement on the code generation task of our proposed multi-agent LLM framework with Deepseek and GPT 3.5 LLM model as the base models. We also conduct comprehensive ablation studies to understand the BlockChain Planner's contribution to our framework. In this project, our proposed multi-agent LLM framework focuses on code generation tasks and we will investigate its effectiveness in other natural language processing tasks in the future to examine its generality ability across different scenarios.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

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*.

Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez. 2023. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. 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, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

R Feurerstein, L Falik, Y Rand, and RS Feurerstein. 2002. The dynamic assessment of cognitive modifiability, icelp, jerusalem.

Linda S Gottfredson. 1997. Mainstream science on intelligence: An editorial with 52 signatories, history, and bibliography.

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*.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*.

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*.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

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

Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2023b. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *arXiv preprint arXiv:2311.05232*.

Yiming Huang, Zhenghao Lin, Xiao Liu, Yeyun Gong, Shuai Lu, Fangyu Lei, Yaobo Liang, Yelong Shen, Chen Lin, Nan Duan, et al. 2023c. Competition-level problems are effective llm evaluators. *arXiv preprint arXiv:2312.02143*.

Lloyd G Humphreys. 1979. The construct of general intelligence.

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

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2023. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *arXiv preprint arXiv:2310.08992*.

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.

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. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. 2023. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*.

Ansh Radhakrishnan, Karina Nguyen, Anna Chen, Carol Chen, Carson Denison, Danny Hernandez, Esin Durmus, Evan Hubinger, Jackson Kernion, Kamilė Lukošiūtė, et al. 2023. Question decomposition improves the faithfulness of model-generated reasoning. *arXiv preprint arXiv:2307.11768*.

Martin Riddell, Ansong Ni, and Arman Cohan. 2024. Quantifying contamination in evaluating code generation capabilities of language models. *arXiv preprint arXiv:2403.04811*.

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

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Robert J Sternberg. 1982. *Handbook of human intelligence*. Cambridge university press.

Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is chatgpt the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938*.

David Wechsler. 1941. The measurement of adult intelligence.

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

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. 2022. Generating sequences by learning to self-correct. *arXiv preprint arXiv:2211.00053*.

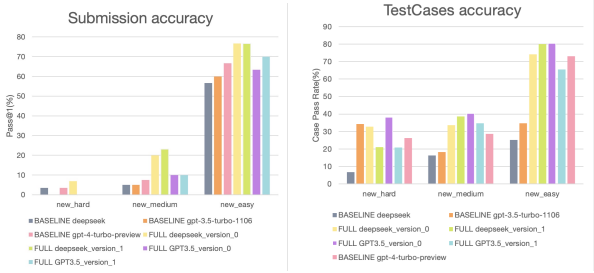
Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Junjie Ye, Xuanting Chen, Nuo Xu, Can Zu, Zekai Shao, Shichun Liu, Yuhan Cui, Zeyang Zhou, Chao Gong, Yang Shen, et al. 2023. A comprehensive capability analysis of gpt-3 and gpt-3.5 series models. *arXiv preprint arXiv:2303.10420*.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*, pages 476–486.

Appendix

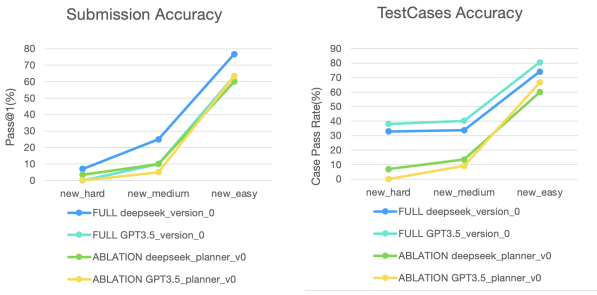


(a) Submission Accuracy New (b) TestCases Accuracy New

Figure 3: Submission Accuracy and TestCases Accuracy for New Problems

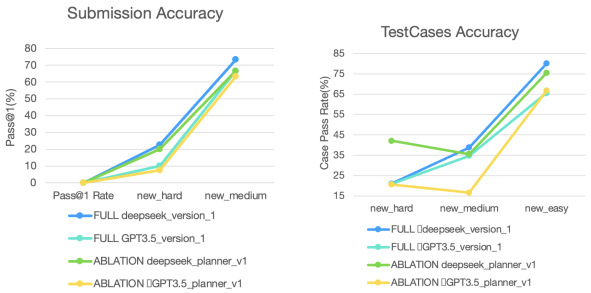
(a) Submission Accuracy Old (b) TestCases Accuracy Old

Figure 4: Submission Accuracy and TestCases Accuracy for Old Problems



(a) Submission Accuracy New (b) TestCases Accuracy New

Figure 5: Submission Accuracy and TestCases Accuracy for New Problems– Ablation study in v0



(a) Submission Accuracy New (b) TestCases Accuracy New

Figure 6: Submission Accuracy and TestCases Accuracy for New Problems– Ablation study in v1

Language Model Code Generation Process

The appendix details the process and output of the LLM as it generates and tests code. Below is the detailed sequence of operations, including code blocks and outputs from the LLM. Note that for the question-based approach we use question 1124 and for the test-based approach we use question 1138. Only two examples are shown here, more examples can be seen on our GitHub repository.

```
============================================================ Calling Self-Reflection Agent ============================================================

------------------------------------------------------------ Enhanced Question Content -------------------------------------------------------------

Objective Summary:
The objective of this problem is to find the length of the longest well-performing interval in a given list of hours worked per day. A well-performing
interval is defined as an interval with more tiring days than non-tiring days.

Variables Included:
- `hours`: A list of integers representing the number of hours worked per day.
- `well-performing interval`: The longest interval with more tiring days than non-tiring days.

Key Constraints and Facts:
- The length of the `hours` list is between 1 and 10^4.
- Each element in the `hours` list is between 0 and 16.
- A day is considered tiring if the number of hours worked is strictly greater than 8.

------------------------------------------------------------------ Hints Content -------------------------------------------------------------------
Insights:
- The problem involves finding the longest well-performing interval, which is an interval with more tiring days than non-tiring days. This insight suggests
that we need to keep track of the number of tiring days and non-tiring days in a sliding window.
- The problem can be solved using a prefix sum and a hash map. The prefix sum can be used to keep track of the number of tiring days up to the current day.
The hash map can be used to store the first occurrence of a certain prefix sum.

Hints:
- To find the longest well-performing interval, we can use a sliding window approach. This hint suggests that we should keep track of the number of tiring
days and non-tiring days in the current window.
- To optimize the solution, we can use a hash map to store the first occurrence of a certain prefix sum. This hint suggests that we should use the hash map
to quickly find the longest well-performing interval.

------------------------------------------------------------ Step-by-Step Plan -------------------------------------------------------------------

Revised step-by-step plan:

1. **Initialization:**
   - Initialize two variables, `tiring_days` and `non_tiring_days`, to 0. These will keep track of the number of tiring and non-tiring days in the current
window.
   - Initialize a hash map, `prefix_sum_index`, to store the first occurrence of a certain prefix sum.
   - Initialize a variable, `max_interval_length`, to 0. This will keep track of the length of the longest well-performing interval found so far.

2. **Iterate over the `hours` list:**
   - For each day, check if the number of hours worked is greater than 8. If it is, increment `tiring_days`, otherwise increment `non_tiring_days`.
   - Calculate the current prefix sum as `tiring_days - non_tiring_days`.
   - Check if the current prefix sum is 0. If it is, the current interval from the start of the list to the current day is a well-performing interval. Update
`max_interval_length` if the current interval is longer.
   - If the current prefix sum is not 0, check if it is already in the hash map. If it is not, add it to the hash map with the current index as the value.
   - If the current prefix sum is already in the hash map, calculate the length of the current interval as the difference between the current index and the
index stored in the hash map. Update `max_interval_length` if the current interval is longer.

3. **Return the result:**
   - After iterating over all days, return `max_interval_length` as the length of the longest well-performing interval.

============================================================================================================================================================
```

Figure 7: Example Output from Question-Based Planner (Plan)

```
============================================================ Calling Coder Agent ============================================================
Here is the Python code that corresponds to the step-by-step plan:

```python
class Solution:
    def longestWPI(self, hours: List[int]) -> int:
        hours = [1 if h > 8 else -1 for h in hours]
        prefix_sum = [0]
        for h in hours:
            prefix_sum.append(prefix_sum[-1] + h)

        stack = []
        for i, s in enumerate(prefix_sum):
            if not stack or prefix_sum[stack[-1]] > s:
                stack.append(i)

        res = 0
        for i in range(len(hours)-1, -1, -1):
            while stack and prefix_sum[stack[-1]] < prefix_sum[i]:
                res = max(res, i - stack.pop())

        return res
```

This code first converts the list of hours into a list of 1s and -1s, where 1 represents a tiring day and -1 represents a non-tiring day. Then it calculates
the prefix sum of this list.

Next, it uses a stack to keep track of the indices of the prefix sum list. The stack is initialized with the first element of the prefix sum list. For each
subsequent element, if it is less than the last element in the stack, it is added to the stack.

Finally, it iterates over the prefix sum list in reverse order. For each element, it pops elements from the stack as long as the current element is greater
than the top element of the stack. The maximum difference between the current index and the index of the top element of the stack is then calculated and
updated in the result.

The time complexity of this code is O(n), where n is the number of elements in the input list. The space complexity is also O(n), as we are using a stack to
store the indices of the prefix sum list.

==
```

Figure 8: Example Output from Question-Based Planner (Code)

```
============================================================ Calling Code Validation Agent ================================================================

------------------------------------------------------------------- Code Solution Analysis -------------------------------------------------------------------
## Critical efficiency assessment and optimisation

### Preliminary evaluation:
- **Necessity of Optimisation**: The current solution is not efficient enough. The time complexity of the current solution is O(n^2) which is not acceptable
for large inputs.
State = True

### Efficiency Observations (if State = True)
- **Algorithmic efficiency**: The current solution uses a brute force approach with a time complexity of O(n^2). This is not efficient for large inputs.

- Data Structure Appropriateness**: The current solution uses a list to store the prefix sum. This is not efficient for large inputs.

### Recommendations (if State = True):
- **Suggestions for streamlined algorithms**: The current solution can be optimized to O(n) using a hash map.

- Data Structure Optimisation**: The current solution uses a list to store the prefix sum. This can be optimized to a hash map.

``````

--- Optimization ---
Optimised_code

```python
def longest_well_performing_interval(hours):
    # Convert hours to tiring (1) and non-tiring (0)
    tiring_days = [1 if h > 8 else 0 for h in hours]
    # Use a dictionary to store the first occurrence of a certain prefix sum
    prefix_sum_index = {0: -1}
    max_length = prefix_sum = 0

    for i, tiring in enumerate(tiring_days):
        # Update prefix sum
        prefix_sum += 1 if tiring else -1
        # If this prefix sum is seen before, there is a well-performing interval
        if prefix_sum in prefix_sum_index:
            max_length = max(max_length, i - prefix_sum_index[prefix_sum])
        else:
            # Otherwise, store the first occurrence of this prefix sum
            prefix_sum_index[prefix_sum] = i

    return max_length
```

This code works by converting the hours worked per day into tiring (1) and non-tiring (0) days. It then uses a dictionary to store the first occurrence of a
certain prefix sum, which represents the number of tiring days minus the number of non-tiring days. If a prefix sum is seen before, it means there is a well-
performing interval between the current day and the day when the prefix sum was first seen. The length of this interval is then compared with the maximum
length found so far.
```

Figure 9: Example Output from Question-Based Optimizer

```
--- Final Optimized Code ---
Test Case Evaluation:

The provided code solution does not have any test cases. However, we can create a set of test cases to evaluate the solution. These test cases should cover a
broad range of scenarios, including boundary conditions, typical cases, and edge cases.

Code Evaluation:

The provided code solution does not have any code evaluation. However, we can perform a code evaluation to identify potential problems. The code seems to be
well-structured and efficient, but it does not have any comments or documentation.

Refined Code Solution:

The provided code solution is already well-structured and efficient. However, it does not have any comments or documentation. We can add comments and
documentation to make the code more understandable.

```python
class Solution:
    def longestWPI(self, hours: List[int]) -> int:
        # Convert hours to tiring (1) and non-tiring (0)
        hours = [1 if h > 8 else 0 for h in hours]

        # Initialize a dictionary to store the first occurrence of a certain prefix sum
        prefix_sum_index = {}
        prefix_sum = 0
        max_length = 0

        # Iterate over the hours list
        for i, h in enumerate(hours):
            # Update the prefix sum
            prefix_sum += 1 if h == 1 else -1

            # If the prefix sum is greater than 0, it means we have a well-performing interval
            if prefix_sum > 0:
                max_length = i + 1

            # If the prefix sum is not in the dictionary, add it
            if prefix_sum not in prefix_sum_index:
                prefix_sum_index[prefix_sum] = i

            # If the prefix sum - 1 is in the dictionary, it means we have a well-performing interval
            if prefix_sum - 1 in prefix_sum_index:
                max_length = max(max_length, i - prefix_sum_index[prefix_sum - 1])

        return max_length
```

State1 = False

The provided code solution does not have any potential problems. It is well-structured and efficient. Therefore, no further optimization is required.

==
```

Figure 10: Example Output from Question-Based Optimizer (Continue)
```

```
========================= Test Output ==============================
● read LeetCode cookies browser=chrome domain=leetcode.com
● running test locally question=longest-well-performing-interval
● Case 1:    Passed
● Case 2:    Passed
Current solution not worse than Best solution so far!

====================================================================
Start Private Tests:
● read LeetCode cookies browser=chrome domain=leetcode.com
● running test locally question=longest-well-performing-interval
● Case 1:    Passed
● Case 2:    Passed
● submitting solution user=xxx@leetcode.com

 √ Accepted

Passed cases:  98/98
Runtime:       169 ms, better than 33%
Memory:        17.2 MB, better than 68%
====================================================================
```

Figure 11: Example Output from Submission (Question-Based)

```
============================================================ Calling Self-Reflection Agent ============================================================
The problem is about reconstructing a 2-row binary matrix given certain constraints.

A binary matrix is a matrix where each element is either 0 or 1. The matrix has 2 rows and 'n' columns, where 'n' is the length of the given array
'colsum'.

The problem provides three pieces of information:

1. `upper`: The sum of elements in the 0-th row of the matrix.
2. `lower`: The sum of elements in the 1-st row of the matrix.
3. `colsum`: An array where each element represents the sum of elements in the corresponding column of the matrix.

The goal is to construct a 2-row binary matrix that satisfies the following conditions:

- The sum of elements in the 0-th row is equal to `upper`.
- The sum of elements in the 1-st row is equal to `lower`.
- The sum of elements in each column is equal to the corresponding element in the `colsum` array.

The matrix should be constructed in such a way that it is valid according to the given constraints. If there are multiple valid solutions, any one of them
is acceptable. If no valid solution exists, an empty 2-D array should be returned.

The constraints on the problem are:

- `1 <= colsum.length <= 10^5`: The length of the `colsum` array should be between 1 and 10^5, inclusive.
- `0 <= upper, lower <= colsum.length`: The sum of elements in the 0-th and 1-st rows should be less than or equal to the length of the `colsum` array.
- `0 <= colsum[i] <= 2`: Each element in the `colsum` array should be between 0 and 2, inclusive.

This problem is asking for a way to construct a binary matrix that meets certain conditions, and it's asking for a solution that is efficient and can
handle large inputs.
```

Figure 12: Example Output from Testcase-Based Planner (Self-reflection)

```
============================================== Calling Testcase-Based Planning Agent ==================================
{'test_case': "Example 3:\n\nInput: upper = 5, lower = 5, colsum = [2,1,2,0,1,0,1,2,0,1]\nOutput: [[1,1,1,0,1,0,0,1,0,0],
[1,0,1,0,0,0,1,1,0,1]]\n\nReflection:\nThe test case is a simple case where the sum of each column is equal to 2. The upper
and lower sums are also equal to 5, which means we need to distribute 5 ones in total. We can distribute them in any way as
long as the sum of each column is 2 and the sum of each row is 5.\n\nThe plan is to first check if it's possible to
construct the matrix according to the given constraints. If not, return an empty matrix. If possible, we will distribute
the ones in a way that maximizes the number of columns with sum 2 and leaves the rest with sum 0 or 1. We will start by
distributing the ones in the columns with sum 2, then distribute the remaining ones in the columns with sum 1.", 'plan':
"1. Define a function `is_possible` to check if it's possible to construct the matrix according to the given
constraints.\n\n2. In `is_possible`, check if the sum of `colsum` is greater than `upper + lower` or if there's a column
with sum 1 but `upper` or `lower` is 0.\n\n3. If `is_possible` returns False, return an empty matrix.\n\n4. If
`is_possible` returns True, define a function `construct_matrix` to construct the matrix.\n\n5. In `construct_matrix`,
initialize the matrix with all zeros.\n\n6. Distribute the ones in the columns with sum 2 first, updating `upper` and
`lower` accordingly.\n\n7. Distribute the remaining ones in the columns with sum 1, updating `upper` and `lower`
accordingly.\n\n8. If `upper` or `lower` is not 0 after distributing all the ones, return an empty matrix.\n\n9. Return the
constructed matrix."}
```

Figure 13: Example Output from Testcase-Based Planner (Plan)

```
================================ Calling Unit Test Agent ================================
Here is the Python code and corresponding unit tests for the given problem:

#### Code Block 1: Construct Matrix
```python
def is_possible(upper, lower, colsum):
    for c in colsum:
        if c == 2:
            upper -= 1
            lower -= 1
        elif c == 1:
            if upper > lower:
                upper -= 1
            else:
                lower -= 1
        elif c > 2:
            return False
    return upper == 0 and lower == 0

def construct_matrix(upper, lower, colsum):
    if not is_possible(upper, lower, colsum):
        return []
    matrix = [[0]*len(colsum) for _ in range(2)]
    for i in range(len(colsum)):
        if colsum[i] == 2:
            matrix[0][i] = matrix[1][i] = 1
            upper -= 1
            lower -= 1
    for i in range(len(colsum)):
        if colsum[i] == 1:
            if upper > 0:
                matrix[0][i] = 1
                upper -= 1
            else:
                matrix[1][i] = 1
                lower -= 1
    return matrix
```

Unit Test for Code Block 1
```python
# Input
upper = 5
lower = 5
colsum = [2,1,2,0,1,0,1,2,0,1]
# Expected Output
expected_matrix = [[1,1,1,0,1,0,0,1,0,0],[1,0,1,0,0,0,1,1,0,1]]
assert construct_matrix(upper, lower, colsum) == expected_matrix
```
```

Figure 14: Example Output from Testcase-Based Planner (Unit Test and Code)

```
============================== Calling Format Agent ==============================
## Complete Function:
```python
class Solution:
    def is_possible(self, upper, lower, colsum):
        for c in colsum:
            if c == 2:
                upper -= 1
                lower -= 1
            elif c == 1:
                if upper > lower:
                    upper -= 1
                else:
                    lower -= 1
            elif c > 2:
                return False
        return upper == 0 and lower == 0

    def construct_matrix(self, upper, lower, colsum):
        if not self.is_possible(upper, lower, colsum):
            return []
        matrix = [[0]*len(colsum) for _ in range(2)]
        for i in range(len(colsum)):
            if colsum[i] == 2:
                matrix[0][i] = matrix[1][i] = 1
                upper -= 1
                lower -= 1
        for i in range(len(colsum)):
            if colsum[i] == 1:
                if upper > 0:
                    matrix[0][i] = 1
                    upper -= 1
                else:
                    matrix[1][i] = 1
                    lower -= 1
        return matrix

    def reconstructMatrix(self, upper: int, lower: int, colsum: List[int]) -> List[List[int]]:
        return self.construct_matrix(upper, lower, colsum)
```
```

Figure 15: Example Output from Testcase-Based Planner (Format)

```
================================ Test Output =================================
● running test locally question=reconstruct-a-2-row-binary-matrix
● Case 1:    Passed
● Case 2:    Passed
● Case 3:    Passed
Current solution not worse than Best solution so far!

============================================================
Start Private Tests:
● running test locally question=reconstruct-a-2-row-binary-matrix
● Case 1:    Passed
● Case 2:    Passed
● Case 3:    Passed
● submitting solution user=wizardly-i3lackburndnq@leetcode.cn

 √ Accepted

Passed cases:  69/69
Runtime:       78 ms, better than 53%
Memory:        21.8 MB, better than 88%
```

Figure 16: Example Output from Submission (Testcase-Based)