A Multi-Agent LLM-Based Architecture for Solving Olympiad-Level Programming Tasks

Anonymous ACL submission

Abstract

We propose a multi-agent architecture that leverages Large Language Models (LLMs) to solve Olympiad-level programming tasks. Our approach takes inspiration from Global Workspace Theory, Society of Mind principles, and has at its core specialized LLM-agents, emphasizing domain-specific adaptation methods like prompt-tuning, fine-tuning (Parameter-Efficient Fine-Tuning, PEFT), retrieval augmented generation. By utilizing smaller, task-focused LLMs over a monolithic model, we enhance accuracy, interpretability, and computational efficiency, providing a scalable strategy for diverse algorithmic challenges.

1 Introduction

The emergence of Large Language Models (LLMs) has revolutionized natural language processing (NLP), enabling significant advancements in reasoning-based tasks such as question answering, code generation(Jiang et al., 2024), and complex problem-solving. However, despite their general reasoning capabilities(Yao et al., 2023), these models face substantial challenges when applied to Olympiad-level programming tasks. Such tasks demand domain-specific expertise, fine-grained interpretability, and efficient problem decomposition(Huang et al., 2024b) — areas where monolithic models often fall short.

To address these limitations, we propose a multi-agent architecture based on specialized LLMs(Ling et al., 2024). Each agent is specilized for a specific class of tasks, collaboratively solving complex problems through a shared workspace. Specialization could be achived by implementing Prompt Tuning techniques(Lester et al., 2021), RAG (Lewis et al., 2021) or training on concerte domain (CT, SFT, PEFT)(Xie et al., 2024; Xu et al., 2023). This approach draws on cognitive frameworks like Global Workspace Theory and the Society of Mind(Zhuge et al., 2023), enabling structured collaboration and enhanced interpretability.

Our architecture focused on open-source LLMs, offering computational efficiency while preserving linguistic versatility. By focusing on NLP-driven applications, we demonstrate how modular, reasoning-capable agents can outperform single large models in tackling diverse algorithmic domains, including Olympiad-level challenges.

2 Related work

The application of LLMs to code generation has gained significant traction, with benchmarks like HumanEval(Chen et al., 2021), CodeContests(Li et al., 2022) and MBPP providing standardized evaluation frameworks (Khan et al., 2023). While these benchmarks reveal the potential of LLMs for producing syntactically correct and semantically meaningful code, they fall short of representing the intricacies of Olympiad-level programming. Current models often struggle with iterative refinement and domain-specific constraints inherent in such tasks.

Techniques like Parameter-Efficient Fine-Tuning (PEFT) (Hu et al., 2021) address these gaps by adapting large, pre-trained LLMs to specialized tasks without incurring excessive computational costs. Adapter-based approaches like LoRA allow selective updates to model parameters, enabling efficient domain adaptation (Lu et al., 2024). In the context of Olympiad programming, this translates to improved domain-specific reasoning while preserving general contextual understanding. However, balancing this specialization with broad applicability remains challenging.

Recent work shows that *retrieval-augmented generation* (RAG) can serve as an effective, low-cost alternative to full fine-tuning when adapting LLMs to specialised problem domains. Instead of storing all domain knowledge in model

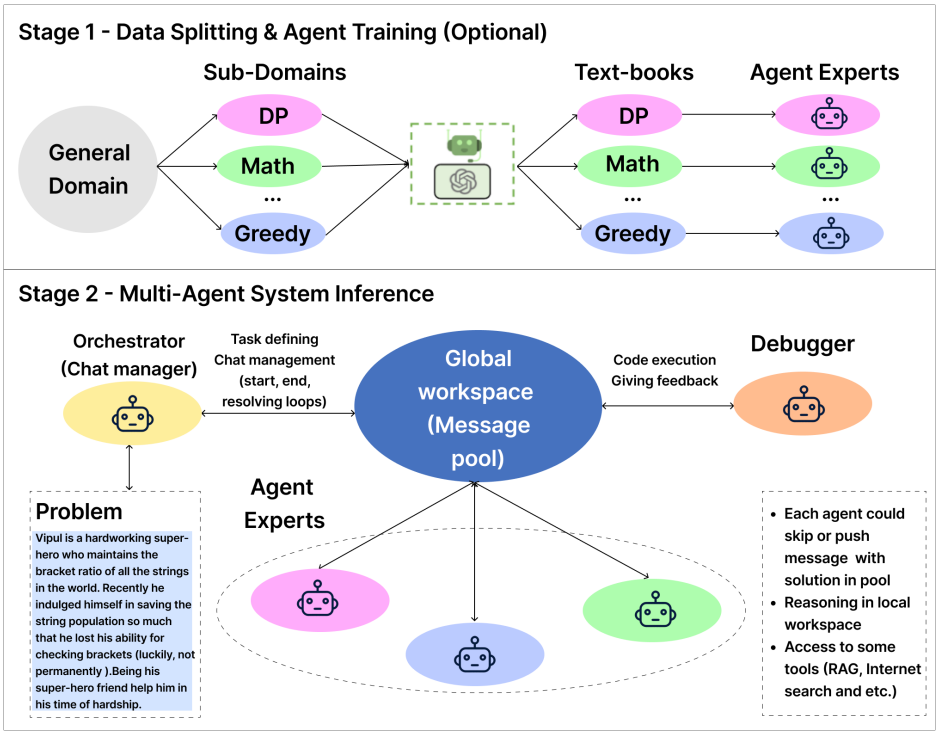

Figure 1: Overview of the proposed multi-agent system architecture. The system operates in two stages: data splitting and agent training, followed by multi-agent system inference.

parameters, RAG attaches a non-parametric memory—typically a vector database of task-specific code snippets, algorithmic templates, and editorial solutions—that is queried at inference time. The retrieved passages are concatenated to the prompt, giving the model up-to-date, highly relevant context and dramatically reducing hallucinations(Lewis et al., 2021; Rohlfs, 2024). For programming contests, domain-focused RAG pipelines have been shown to boost pass@k on HumanEval and CodeContests without increasing model size, because reusable patterns such as DP recurrences or number-theoretic lemmas can be surfaced on demand(Wang et al., 2023; Parvez et al., 2021). Moreover, the non-parametric store can be refreshed continuously with new problems and editorial analyses, enabling rapid adaptation to evolving contest styles while preserving the base model's general language competence.

The multi-agent paradigm offers a promising solution by combining the capabilities of specialized models within a shared framework. Inspired by cognitive theories like Society of Mind (Smoliar, 1991) and Global Workspace Theory, this approach enables information exchange, error-correction, and adaptive task allocation among agents. Recent implementations such as AgentCoder(Huang et al., 2024a) and MapCoder(Islam et al., 2024) demonstrate the advantages of modular systems, achieving improved accuracy and interpretability by dividing tasks into sub-problems like parsing, code generation, and testing.

Despite their success, multi-agent systems face obstacles in agent coordination, redundancy, and dynamic task-switching across diverse problem domains. Our work builds on these foundations, employing domain adoptated LLM agents within a shared architecture that optimizes resource allocation and ensures robust inter-agent communication. We demonstrate the effectiveness of this approach for Olympiad-level programming tasks, emphasizing its relevance to NLP-driven solutions.

3 Proposed Method

The proposed method consists of two main stages: agent training and inference within a multi-agent framework. This approach is designed to optimize memory efficiency and computational performance while solving Olympiad-level programming tasks.

3.1 Stage 1: Data Splitting and Agent Training

In the first stage, agents are trained for specific sub-domains such as dynamic programming (DP), mathematics, and greedy algorithms. This stage is optional, we could use other mentioned domain adoptation techniques. The training process follows these steps:

- **Data Preparation**: A general dataset is divided into sub-domain-specific tasks. These

tasks are augmented with textbook-style examples that represent typical problems within each sub-domain.

- **Fine-Tuning**: Agents are fine-tuned using adapter-based techniques like LoRA, which allow for lightweight parameter updates without retraining the base model.

3.2 Stage 2: Multi-Agent System Inference

In the inference stage, a multi-agent system, built on the AutoGen(Wu et al., 2023) framework, collaborates to solve complex tasks. The process is as follows:

- **Orchestrator**: Acts as the central controller, managing task allocation, coordinating agent activities, and resolving iterative loops.
- **Global Workspace**: A shared message pool inspired by Global Workspace Theory, where agents exchange solutions and feedback.
- **Agent Experts**: Specialized agents work iteratively, contributing solutions based on their domain expertise. Each agent specilized by Prompt-Tuning or LoRA adapter attached to the shared base model.
- **Debugger**: Validates proposed solutions, executes code, and provides feedback to refine results.

4 Experiments

4.1 Domain Adaptation Setup

To equip every agent with reliable, fine-grained expertise we apply a three-layer adaptation pipeline that combines *prompt-tuning*, *retrieval-augmented generation* (RAG), and *parameter-efficient fine-tuning*.

Prompt-Tuning. During inference the orchestrator pre-appends a compact system prompt defining the agent's role (*"You are a specialist in dynamic programming. . . "*), listing canonical solution templates, and specifying the desired answer format. This zero-parameter step yields immediate domain alignment and graceful fallback when external resources are unavailable.

Retrieval-Augmented Generation. Next, the prompt is extended with passages retrieved from a curated knowledge store dedicated to the active algorithmic domain. Each store is indexed with FAISS and contains competitive-programming wiki pages, textbook summaries, and hand-written cheat-sheets. Given a problem statement, the retriever returns the top-k (we use k=5) snippets that surface reusable recurrences, proof sketches, or reference implementations, lowering hallucination rates while keeping the base model unchanged.

Parameter-Efficient Fine-Tuning. Finally, we train a LoRA adapter per domain on a synthetic corpus of problem–solution pairs generated by self-play with Qwen-2.5-72B and filtered through unit tests. Adapters are mounted on the shared Phi-4-14B backbone, updating fewer than 1% of its weights.

4.2 Inference Setup

The inference system is implemented using the AutoGen framework, with vLLM providing optimized memory and computational handling. Key aspects include:

- **Dynamic Adapter Loading**: During inference, the system maintains a single shared base model and dynamically loads LoRA adapters corresponding to active agents.
- **Task Orchestration**: The orchestrator ensures efficient task allocation and coordination among agents, minimizing redundant computations.
- **Efficient Execution**: Debugging and validation are performed iteratively, guided by the orchestrator and global workspace.

5 Results

5.1 Domain adoptation

Method	Python	C++17
phi-4 baseline	4.5% (2/44)	22.27% (10/44)
phi-4 + PE (Specialisation)	9% (4/44)	22.27% (11/44)
phi-4 + RAG	–	–
phi-4 SFT (LoRA)	13.63% (6/44)	22.27% (11/44)

Table 1: pass@10 on *dynamic_programming* tasks (44 problems).

Method	Python	C++17
phi-4 baseline	9.67% (3/31)	25.8% (8/31)
phi-4 + PE (Specialisation)	9.67% (3/31)	25.8% (8/31)
phi-4 + RAG	–	–
phi-4 SFT (LoRA)	12.9% (4/31)	25.8% (8/31)

Table 2: pass@10 on *graphs* tasks (31 problems).

Discussion. Prompt-engineering specialisation (row 2) improves Python accuracy on dynamic-programming tasks compared with the baseline, confirming that lightweight domain cues can help the model focus on the right solution space. However, neither PEFT nor LoRA adapters alone yield convincing gains across languages; in fact, the PEFT-specialised C++17 scores are virtually identical to the baseline. The LoRA adapter brings a further ≈+4 pp boost on dynamic-programming (Python) but still under-performs the baseline in C++. Overall, the numbers suggest that domain adaptation by PEFT or prompt-tuning in isolation is inadequate for highly compositional reasoning; richer external context (RAG) or training regimes that encourage step-wise reasoning are likely required.

6 Conclusion

7 Limitations

Our study omits two costly yet promising research directions: (i) reinforcement-learning-based fine-tuning (e.g., GRPO or RLHF) and (ii) the use of dedicated reasoning-augmented backbones. Both demand significant GPU budgets beyond the scope of this work.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *Preprint*, arXiv:2106.09685.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024a. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *Preprint*, arXiv:2312.13010.

Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. 2024b. Understanding the planning of llm agents: A survey. *Preprint*, arXiv:2402.02716.

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

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *Preprint*, arXiv:2406.00515.

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

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. *Preprint*, arXiv:2104.08691.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Preprint*, arXiv:2005.11401.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Chen Ling, Xujiang Zhao, Jiaying Lu, Chengyuan Deng, Can Zheng, Junxiang Wang, Tanmoy Chowdhury, Yun Li, Hejie Cui, Xuchao Zhang, Tianjiao Zhao, Amit Panalkar, Dhagash Mehta, Stefano Pasquali, Wei Cheng, Haoyu Wang, Yanchi Liu, Zhengzhang Chen, Haifeng Chen, Chris White, Quanquan Gu, Jian Pei, Carl Yang, and Liang Zhao. 2024. Domain specialization as the key to make large language models disruptive: A comprehensive survey. *Preprint*, arXiv:2305.18703.

Wei Lu, Rachel K. Luu, and Markus J. Buehler. 2024. Fine-tuning large language models for domain adaptation: Exploration of training strategies, scaling, model merging and synergistic capabilities. *Preprint*, arXiv:2409.03444.

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

Chris Rohlfs. 2024. Generalization in neural networks: A broad survey. *Preprint*, arXiv:2209.01610.

Stephen W. Smoliar. 1991. The society of mind: Marvin minsky. *Artificial Intelligence*, 48(3):349–370.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *Preprint*, arXiv:2305.07922.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation. *Preprint*, arXiv:2308.08155.

Yong Xie, Karan Aggarwal, and Aitzaz Ahmad. 2024. Efficient continual pre-training for building domain specific large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 10184–10201, Bangkok, Thailand. Association for Computational Linguistics.

Lingling Xu, Haoran Xie, Si-Zhao Joe Qin, Xiaohui Tao, and Fu Lee Wang. 2023. Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment. *Preprint*, arXiv:2312.12148.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

Mingchen Zhuge, Haozhe Liu, Francesco Faccio, Dylan R. Ashley, Róbert Csordás, Anand Gopalakrishnan, Abdullah Hamdi, Hasan Abed Al Kader Hammoud, Vincent Herrmann, Kazuki Irie, Louis Kirsch, Bing Li, Guohao Li, Shuming Liu, Jinjie Mai, Piotr Piękos, Aditya Ramesh, Imanol Schlag, Weimin Shi, Aleksandar Stanić, Wenyi Wang, Yuhui Wang, Mengmeng Xu, Deng-Ping Fan, Bernard Ghanem, and Jürgen Schmidhuber. 2023. Mindstorms in natural language-based societies of mind. *Preprint*, arXiv:2305.17066.

A Appendix

Instruction

Recently Ivan the Fool decided to become smarter and study the probability theory. He thinks that he understands the subject fairly well, and so he began to behave like he already got PhD in that area.\n\nTo prove his skills, Ivan decided to demonstrate his friends a concept of random picture. A picture is a field of n rows and m columns, where each cell is either black or white. Ivan calls the picture random if for every cell it has at most one adjacent cell of the same color. Two cells are considered adjacent if they share a side.\n\nIvan's brothers spent some time trying to explain that it's not how the randomness usually works. Trying to convince Ivan, they want to count the number of different random (according to Ivan) pictures. Two pictures are considered different if at least one cell on those two picture is colored differently. Since the number of such pictures may be quite large, print it modulo 10^9 + 7.
Input
The only line contains two integers n and m (1 ≤ n, m ≤ 100 000), the number of rows and the number of columns of the field.
Output
Print one integer, the number of random pictures modulo 10^9 + 7.
Example
Input 2 3 Output 8
Note
The picture below shows all possible random pictures of size 2 by 3.

Solution

To solve this problem, we can use dynamic programming to efficiently compute the number of random pictures according to Ivan's definition. The main idea is to calculate the number of valid configurations for a single row and then use this to compute the total number of valid configurations for the entire picture. Here's how we can approach the problem:

1. **Define the DP Array**:
 - Let `a[i]` be the number of valid configurations for a row of length `i`.

2. **Initialize**:
 - Set `a[0] = 0`, `a[1] = 2`, and `a[2] = 4` as base cases for the number of valid configurations for rows of length 0, 1, and 2 respectively.

3. **Compute the DP Array**:
 - For each position `i` from 3 to the maximum of `n` and `m`, calculate `a[i]` as the sum of `a[i-1]` and `a[i-2]` modulo `10^9 + 7`. This is because a valid configuration for a row of length `i` can be formed by appending a valid configuration of length `i-1` or `i-2` with a valid ending.

4. **Calculate the Total Number of Pictures**:
- The total number of random pictures is the sum of the number of valid configurations for a row of length `m` minus 2 (to exclude the configurations that are not valid for the first row) and the number of valid configurations for a row of length `n`.

5. **Modulo Operation**:
- Since the answer can be large, take modulo `10^9 + 7` at each step.

Here the code to solve the problem:
```python
n, m = map(int, input().split())
mod = 10**9+7
a = [0,2,4]
for i in range(3, max(n, m)+1):
  a.append((a[i-1]+a[i-2])%mod)
print((a[m]-2 + a[n])%mod)```

Figure 2: Textbook example
```