A Formalization of Multi-Agent Interaction

Many studies adopt Partially Observable Stochastic Games (POSG) to model the LLM interaction in MAS Slumbers et al. [2024], Park et al. [2025], Liu et al. [2025], Sarkar et al. [2025]. In this section, we show that Dec-POMDP offers special merits compared to POSG in the solution concept in the cooperative settings, thus more suited to model LLM collaboration.

A.1 Dec-POMDP

A Dec-POMDP is defined by $\langle \mathcal{I}, \mathcal{S}, \{\mathcal{O}_i\}, \{\mathcal{A}_i\}, R, T, H \rangle$. At each step t, since an agent cannot directly observe the state s_t, it usually maintains local observation-action history $h_{i,t} = (o_{i,0}, a_{i,0}, \ldots, o_{i,t})$ to infer a belief over the underlying state. Decisions are made according to a local policy $\pi_i : \mathcal{H}_{i,t} \to \Delta(\mathcal{A}_i)$, which maps histories to probability distributions over actions. The set of all local policies forms the joint policy $\boldsymbol{\pi} = \{\pi_1, \ldots, \pi_n\}$. In cooperative settings, the objective is to maximize shared cumulative rewards. As proved in Oliehoek et al. [2008], there is always an optimal joint policy in a Dec-POMDP,

$$\boldsymbol{\pi}^* = \underset{\boldsymbol{\pi} \in \boldsymbol{\Pi}}{\operatorname{argmax}} \, \mathbb{E}_{\boldsymbol{\pi}} \left[\sum_{t=0}^{H-1} R(s_t, a_t) \right]. \tag{1}$$

A.2 POSG

A Partially Observable Stochastic Game (POSG), so-called Partially Observable Markov Game (POMG), does not assume cooperative behavior among agents. It can be either a cooperative, competitive, or mixed game. A POSG is defined as $\langle \mathcal{I}, \mathcal{S}, \{\mathcal{A}_i\}, T, \{\mathcal{O}_i\}, O, \{R_i\}, H \rangle$, where each agent has its own reward function $R_i : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. In POSG, each agent seeks to maximize its individual return under the fixed policies of all others $\boldsymbol{\pi}_{-i}$. The optimal policy $\pi_i^{\circledast}$ for each agent $i \in \mathcal{I}$ is,

$$\pi_i^{\circledast} = \underset{\pi_i \in \Pi_i}{\operatorname{argmax}} \, \mathbb{E}_{\pi_i, \boldsymbol{\pi}_{-i}} \left[\sum_{t=0}^{H-1} R_i(s_t, a_t) \right], \tag{2}$$

The solutions for POSG are Nash Equilibria (NE), where no agents can unilaterally improve their returns by deviating from their policies. Formally, for all $i \in \mathcal{I}$ and any alternative policy $\pi_i \in \Pi_i$, NE satisfy

$$\mathbb{E} \left[\sum_{t=0}^{H-1} R_i(s_t, a_t) \mid \pi_i^{\circledast}, \boldsymbol{\pi}_{-i}^{\circledast} \right] \geq \mathbb{E} \left[\sum_{t=0}^{H-1} R_i(s_t, a_t) \mid \pi_i, \boldsymbol{\pi}_{-i}^{\circledast} \right]. \tag{3}$$

Like Dec-POMDP, the decision-making in POSG is still concurrent (as stochastic games), where all agents act synchronously at each time step. In contrast, turn-based interactions, where agents take turns to act (e.g., chess, Kuhn Poker, tic-tac-toe), are typically modeled as extensive-form games.

A.3 Non-Optimality of POSG Solutions

We illustrate that the solutions of POSG, i.e., NE, may not necessarily lead to joint optimality in cooperative settings.

Consider a one-step matrix game involving 2 agents, where each agent selects an action from the action space $\mathcal{A} = \{\mathcal{A}^{(1)}, \mathcal{A}^{(2)}\}$. The joint action profile determines the utility as presented in Table 1.

$a_1 \backslash a_2$	$\mathcal{A}^{(1)}$	$\mathcal{A}^{(2)}$
$\mathcal{A}^{(1)}$	10	7
$\mathcal{A}^{(2)}$	7	0

Table 1: Joint utility matrix of 2 agents.

This matrix game can be potentially decomposed into 2 POSG in Table 2 through reward shaping.

$a_1 \backslash a_2$	$\mathcal{A}^{(1)}$	$\mathcal{A}^{(2)}$
$\mathcal{A}^{(1)}$	(5, 5)	(3, 4)
$\mathcal{A}^{(2)}$	(4, 3)	(0, 0)

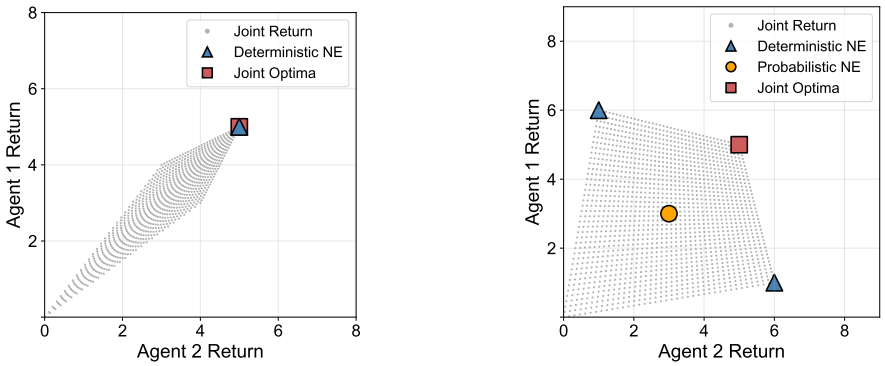
(a) POSG 1

$a_1 \backslash a_2$	$\mathcal{A}^{(1)}$	$\mathcal{A}^{(2)}$
$\mathcal{A}^{(1)}$	(5, 5)	(1, 6)
$\mathcal{A}^{(2)}$	(6, 1)	(0, 0)

(b) POSG 2

Table 2: Return tables of 2 POSG.

In the POSG presented in Table 2a, $(\mathcal{A}^{(1)}, \mathcal{A}^{(1)})$ is a Nash equilibrium (blue triangle in Figure 1a). When $a_1 = \mathcal{A}^{(1)}$, $U_2(\mathcal{A}^{(1)}, \mathcal{A}^{(1)}) > U_2(\mathcal{A}^{(1)}, \mathcal{A}^{(2)})$; when $a_1 = \mathcal{A}^{(2)}$, $U_2(\mathcal{A}^{(2)}, \mathcal{A}^{(1)}) > U_2(\mathcal{A}^{(2)}, \mathcal{A}^{(2)})$. Therefore, the best response for agent 2 is $a_2^{\circledast} = \mathcal{A}^{(1)}$. Similarly, since $U_1(\mathcal{A}^{(1)}, \mathcal{A}^{(1)}) > U_1(\mathcal{A}^{(2)}, \mathcal{A}^{(1)})$, we obtain $a_1^{\circledast} = \mathcal{A}^{(1)}$. This NE also achieves joint optimality with the maximum utility $5 + 5 = 10$ (red square in Figure 1a).



(a) POSG 1 (b) POSG 2

Figure 1: Utility spaces of 2 POSG.

However, certain reward decompositions may yield non-optimal solutions for cooperative games in Table 2, even when POSG solutions reach NE. For the POSG shown in Table 2b, the deterministic NE are $(\mathcal{A}^{(1)}, \mathcal{A}^{(2)}), (\mathcal{A}^{(2)}, \mathcal{A}^{(1)})$ (blue triangles in Figure 1b). When $a_1 = \mathcal{A}^{(1)}$, agent 2 prefers $\mathcal{A}^{(2)}$ as $U_2(\mathcal{A}^{(1)}, \mathcal{A}^{(2)}) > U_2(\mathcal{A}^{(1)}, \mathcal{A}^{(1)})$; when $a_1 = \mathcal{A}^{(2)}$, agent 2 prefers $\mathcal{A}^{(1)}$ since $U_2(\mathcal{A}^{(2)}, \mathcal{A}^{(1)}) > U_2(\mathcal{A}^{(2)}, \mathcal{A}^{(2)})$. Agent 1 faces the same issue. Thus, neither agent can unilaterally improve their utilities by deviating. However, the collective utilities obtained from both policies yield $6 + 1 = 7 < 10$, which are suboptimal compared to the joint optimum (red square in Figure 1b).

In Table 2b, even the probabilistic NE under stochastic policies is still non-optimal. Suppose agent 1 selects $\mathcal{A}^{(1)}$ with probability p, and agent 2 selects $\mathcal{A}^{(1)}$ with probability q, $R_1(\mathcal{A}^{(1)}, \cdot) = 5q + (1 - q) = 4q + 1$, $R_1(\mathcal{A}^{(2)}, \cdot) = 6q$, $R_1(\mathcal{A}^{(1)}, \cdot) = R_1\mathcal{A}^{(2)}, \cdot)$ yields $q = 0.5$; similarly, $R_2(\cdot, \mathcal{A}^{(1)}) = 5p + (1 - p) = 4p + 1$, $R_2(\cdot, \mathcal{A}^{(2)}) = 6p$, $R_2(\mathcal{A}^{(1)}, \cdot) = R_2\mathcal{A}^{(2)}, \cdot)$ yields $p = 0.5$. This probabilistic NE, $\pi_1^{\circledast}(\mathcal{A}^{(1)}) = \pi_1^{\circledast}(\mathcal{A}^{(2)})$, $\pi_2^{\circledast}(\mathcal{A}^{(1)}) = \pi_2^{\circledast}(\mathcal{A}^{(2)})$ leads to overall utilities $3 + 3 = 6 < 10$ (orange circle in Figure 1b).

Although appropriate reward shaping techniques can transform a cooperative game into a POSG like Table 2a to make the NE also jointly optimal, this becomes more challenging when more agents are involved and episodes become longer. We employ Dec-POMDP to avoid the intricate reward engineering and seek the joint optimality.

B Cooperation Schemes

By training the auxiliary and main coders to cooperate under minimal constraints (with only the problem description and their respective roles provided), diverse cooperation schemes naturally emerge. We present 4 representative schemes observed in our models.

B.1 Fallback

The most commonly observed cooperation scheme is the main agent providing a fallback for the auxiliary function. Although prompted to use the auxiliary output, the main agent may still doubt its completeness and reimplement the solution as a backup in case of potential errors.

Listing 1: `compare_one(a,b)`

```
def aux(x, y):
    # Convert x to a float
    if isinstance(x, str):
        x = float(x.replace(",", "."))

    # Vulnerable if y in different types
    if x < y:
        return y
    else:
        return x

def compare_one(a, b):
    try:
      return aux(a, b)
  except Exception:
      # Robust implementation
      if type(a) == type(b):
          if a > b:
                return a
          elif a < b:
                return b
          else:
                return None
      elif type(a) == float and type(b) == str:
          b_float = float(b.replace(",", "."))
          if a > b_float:
                return a
          elif a < b_float:
                return b_float
          else:
                return None
      elif type(a) == str and type(b) == float:
            a_str = str(a.replace(",", "."))
            a_float = float(a_str)
          if a_float > b:
                return a_str
          elif a_float < b:
                return b
          else:
                return None
      else:
          return None
```

For example, the problem `compare_one(a,b)` asks to return the larger of 2 values in its original type (string, float, or int), or `None` if equal. Listing 1 demonstrates how the fallback mechanism provides an alternative in case of runtime errors in the auxiliary implementation. The `aux(x,y)` function converts variable `x` to a float, resulting in an operand type mismatch if `y` is a string. The main agent handles the type error by falling back to its secondary implementation. While the main implementation may still be vulnerable, this mechanism benefits small models that commonly encounter syntax and runtime errors. Despite some redundancy, a proper fallback can enhance code robustness.

B.2 Decorator

When the main agent trusts the auxiliary to accurately handle the core logic, it typically acts as a decorator, adding complementary features to enhance the output. The main agent could handle edge cases for improved robustness or reformat the outputs (e.g., sorting, rounding, and adjusting units). The necessity of such decorating logic depends on the auxiliary agent's functionality and completeness.

Listing 2: `is_palindrome(text)`

```
def aux1(text):
    if len(text) == 1:
        return True
    else:
        return aux1(text[1:-1])

def aux2(text):
    if len(text) <= 1:
        return True
    else:
        return aux2(text[1:-1])

def is_palindrome(text):
    if not text:
        return True

    # Edge case handler
    return text[0] == text[-1] and aux1(text)

    # Redundant decorator
    return text[0] == text[-1] and aux2(text)
```

Listing 2 illustrates a decorator main function and how its utility varies depending on the completeness of the auxiliary implementation. In this problem, 2 agents generate codes together to check if a given string is a palindrome. When paired with `aux1` that only handles the recursion boundary condition of single-character strings, the empty string check of the main serves as a necessary edge case handler. However, when working with `aux2`, which already has a more comprehensive edge case consideration, this handle becomes redundant.

B.3 Coordinator

In large-scale software systems, it would be beneficial to have pipelines for repeated or data-parallel operations (e.g., batch processing, stream transformations). This corresponds to the coordinator cooperation scheme in our models, where the main agent divides the tasks into parts and assigns them to the auxiliary agent.

A simple example involves the main agent acting as an iterator, using a loop (e.g., a `for` loop) to structure the problem. The auxiliary function generates partial solutions within each iteration. These partial results are then aggregated to form the final output. However, this cooperation scheme is unstable, as it depends heavily on the correct functionality of the auxiliary agent. When the auxiliary agent fails to complete its subtask, the entire solution breaks down.

Listing 3: `flip_case(string)`

```
def aux(string: str) -> str:
    result = ""
    for char in string:
        if char.islower():
            result += char.upper()
        elif char.isupper():
            result += char.lower()
        else:
            result += char
    return result

def flip_case(string: str):
    flipped = ""
    for char in string:
        flipped += aux(char)
    return flipped
```

Listing 3 demonstrates a solution to flip the case of characters in a string. The auxiliary function flips the case of each character, while the main function calls this auxiliary function for each character and appends it to the result. This scheme can be extended to more complex scenarios, where subtasks are assigned in a hierarchical structure.

B.4 Strategy Filter

When handling complex problems, the main agent may need to implement logic based on multiple conditions. In such cases, the auxiliary agent can act as a filter for specific branches of logic, often appearing within conditional blocks (e.g., following an `if` statement). This scheme resembles the adaptive control flow in practice. In rule-based pipelines, an auxiliary agent evaluates preconditions (e.g., task types, system status, configurations) and directs workers to execute appropriate subroutines, thereby enhancing project modularity.

Listing 4: `x_or_y(n,x,y)`

```
def aux(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(3, int(n**0.5) + 1, 2):
        if n % i == 0:
            return False
    return True

def x_or_y(n, x, y):
    # Check if n is prime
    if aux(n):
        return x
    else:
        return y
```

Listing 4 presents a solution for `x_or_y(n,x,y)` problem, which returns `x` if `n` is prime and `y` otherwise. The auxiliary function handles the primality checking, while the main function is responsible for returning results. The same pattern can also be found in the solutions of `prime_fib(n)`, `factorize(n)`, and `largest_prime_factor(n)`.

C Broader Impacts

Prompt-based coordination is often brittle Estornell and Liu [2024], as agents may fail to follow instructions they were not explicitly trained to interpret. Our method builds on a solid theoretical foundation in cooperative MARL, explicitly optimizing agents for joint optimality. Our work also opens opportunities to enhance existing test-time multi-agent interaction methods by integrating MARL techniques Du et al. [2023], Lifshitz et al. [2025], Wu et al. [2023a], particularly in settings that involve task decomposition and iterative feedback integration.

This work also explores a new perspective on accelerating LLM inference through cooperative MARL. While mainstream acceleration techniques (e.g., knowledge distillation, pruning, and quantization) improve efficiency at the cost of information loss Wang et al. [2024], Zhao et al. [2024], our approach suggests decentralized coordination among specialized agents, thereby alleviating the burden of long-context memory and joint decision-making on a single model. Each agent can focus on a specific subtask, enabling more modular and robust reasoning.

D Limitations and Future Works

Nevertheless, this study is subject to several limitations. First, we focus on homogeneous agents for simplicity, assuming they perform similar tasks despite being assigned different roles, e.g., both the auxiliary agent and main agent are generating Python functions. Future research could explore LLM collaboration among heterogeneous agents with diverse capabilities and functionalities.

Due to computational constraints, we train LLMs with MAGRPO on limited datasets using relatively small-scale language models. When LLM-based coding agents are deployed in larger-scale projects involving multiple files and modules, more diverse and complex cooperation schemes are likely to emerge, which would further demonstrate the potential of decentralized coordination in MAS.

The simplicity of our reward model inevitably leads to narrow reward signals and potential reward hacking. As suggested by many research studies and industrial practice Uesato et al. [2022], Wu et al. [2023b], designing more expressive and fine-grained reward models (e.g., multi-aspect rewards, process-supervised rewards) is essential for better aligning agent cooperation with human preferences.

References

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate, 2023. URL https://arxiv.org/abs/2305.14325.

Andrew Estornell and Yang Liu. Multi-llm debate: Framework, principals, and interventions. In *Neural Information Processing Systems (NeurIPS)*, 2024. URL https://openreview.net/forum?id=sy7eSEXdPC.

Shalev Lifshitz, Sheila A. McIlraith, and Yilun Du. Multi-agent verification: Scaling test-time compute with multiple verifiers, 2025. URL https://arxiv.org/abs/2502.20379.

Bo Liu, Leon Guertler, Simon Yu, Zichen Liu, Penghui Qi, Daniel Balcells, Mickel Liu, Cheston Tan, Weiyan Shi, Min Lin, Wee Sun Lee, and Natasha Jaques. Spiral: Self-play on zero-sum games incentivizes reasoning via multi-agent multi-turn reinforcement learning, 2025. URL https://arxiv.org/abs/2506.24119.

F. A. Oliehoek, M. T. J. Spaan, and N. Vlassis. Optimal and approximate q-value functions for decentralized pomdps. *Journal of Artificial Intelligence Research*, 32:289–353, May 2008. ISSN 1076-9757. doi: 10.1613/jair.2447. URL http://dx.doi.org/10.1613/jair.2447.

Chanwoo Park, Seungju Han, Xingzhi Guo, Asuman Ozdaglar, Kaiqing Zhang, and Joo-Kyung Kim. Maporl: Multi-agent post-co-training for collaborative large language models with reinforcement learning, 2025. URL https://arxiv.org/abs/2502.18439.

Bidipta Sarkar, Warren Xia, C. Karen Liu, and Dorsa Sadigh. Training language models for social deduction with multi-agent reinforcement learning, 2025. URL https://arxiv.org/abs/2502.06060.

Oliver Slumbers, David Henry Mguni, Kun Shao, and Jun Wang. Leveraging large language models for optimised coordination in textual multi-agent reinforcement learning, 2024. URL https://openreview.net/forum?id=1PPjf4wife.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process- and outcome-based feedback, 2022. URL https://arxiv.org/abs/2211.14275.

Bichen Wang, Yuzhe Zi, Yixin Sun, Yanyan Zhao, and Bing Qin. Rkld: Reverse kl-divergence-based knowledge distillation for unlearning personal information in large language models, 2024. URL https://arxiv.org/abs/2406.01983.

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. Autogen: Enabling next-gen llm applications via multi-agent conversation, 2023a. URL https://arxiv.org/abs/2308.08155.

Zeqiu Wu, Yushi Hu, Weijia Shi, Nouha Dziri, Alane Suhr, Prithviraj Ammanabrolu, Noah A. Smith, Mari Ostendorf, and Hannaneh Hajishirzi. Fine-grained human feedback gives better rewards for language model training, 2023b. URL https://arxiv.org/abs/2306.01693.

Pu Zhao, Fei Sun, Xuan Shen, Pinrui Yu, Zhenglun Kong, Yanzhi Wang, and Xue Lin. Pruning foundation models for high accuracy without retraining, 2024. URL https://arxiv.org/abs/2410.15567.