# Can LLMs help in Multi Agent environment ?

May 12, 2024


## Authors

- Rohan Kumar (2021101070)
- Ashutosh Srivastava (2021101056)

# Contents

# 1 LLMs for Multi Agent Reinforcement Learning

## 1.1 Introduction

LLMs possess vast amounts of general world knowledge and language understanding capabilities that can be beneficial for MARL agents. By integrating LLMs into the MARL framework, agents can tap into this rich knowledge to better understand their environment, communicate more effectively with other agents, and make more informed decisions.

One key approach is to "align" the LLM with the functional knowledge required for the cooperative MARL task through a centralized on-policy update rule . This allows the LLM to develop an understanding of the task-specific coordination required, rather than relying solely on its general knowledge.

Additionally, LLMs can be used to enable more natural and intuitive communication between agents, leveraging their linguistic strengths for message passing. This can lead to more effective coordination strategies emerging, as agents are able to convey complex ideas and intentions to their teammates.

By combining the reasoning and planning capabilities of RL with the broad knowledge and communication abilities of LLMs, MARL agents can learn to cooperate and solve tasks more efficiently than using RL alone. This can lead to faster generalization across different environments and tasks, as the agents can draw upon the LLM's prior knowledge rather than having to learn everything from scratch.

The integration of LLMs into MARL setups is a promising direction for developing more capable and coordinated multi-agent systems.

## 1.2 Central Problem

The central problem in the integration of Large Language Models (LLMs) into Multi-Agent Reinforcement Learning (MARL) setups is the challenge of converting LLM responses to actions or policy updates for the agents in the multi-agent system.

LLMs possess vast language understanding capabilities and general world knowledge, which can be valuable for MARL agents. However, effectively utilizing this knowledge to improve the performance of MARL agents requires converting the information provided by the LLM into actionable decisions or policy updates within the MARL framework.

This involves several sub-challenges:

1. Semantic Alignment: Ensuring that the information provided by the LLM is semantically aligned with the objectives and tasks of the MARL environment. The responses from the LLM must be relevant and meaningful in the context of the cooperative tasks the agents are attempting to solve.
2. Actionable Insights: Converting the high-level knowledge and insights provided by the LLM into actionable decisions or policy updates for the individual agents. This requires mapping the language-based responses from the LLM to specific actions or coordination strategies that the agents can implement.
3. Dynamic Adaptation: Developing mechanisms to dynamically adapt the LLM responses based on the changing environment and the interactions between agents. The LLM's responses may need to be updated or refined over time as the agents learn and the task requirements evolve.
4. Efficient Communication: Ensuring that the communication between the LLM and the agents is efficient and effective. This may involve designing protocols or interfaces for exchanging information between the LLM and the MARL framework, as well as optimizing the communication process to minimize overhead and latency.

Addressing these challenges requires interdisciplinary research at the intersection of natural language processing, reinforcement learning, and multi-agent systems. By developing techniques to effectively integrate LLMs into MARL setups, we can unlock the potential for more capable and coordinated multi-agent systems that leverage the power of language understanding to solve complex tasks.

# 2 Literature Review

## 2.1 Enabling Intelligent Interactions between an Agent and an LLM: A Reinforcement Learning Approach

The paper introduces When2Ask, a reinforcement learning framework that optimizes agent interactions with large language models (LLMs). The aim is to minimize the interaction cost while maintaining the agent's performance in complex decision-making tasks.

Key Contributions:

1. Problem Statement and Solution: The authors frame the challenge as a Markov decision process (MDP) and introduce a Planner-Actor-Mediator architecture where the mediator learns a policy to decide when to query an LLM for high-level guidance.
2. Interaction Management: When2Ask leverages reinforcement learning (RL) to dynamically learn when to ask an LLM for instructions versus continuing with the existing plan.
3. Environment Testing: The approach is evaluated in MiniGrid and Habitat environments, where When2Ask achieves high task success rates with a significant reduction in LLM interactions compared to baseline methods.
4. Robust Performance: The mediator model ensures that the agent remains robust against partial observability by proactively requesting guidance only when needed.

Results:

When2Ask outperforms other baseline strategies like always querying the LLM or using hard-coded rules, demonstrating both resource efficiency and effectiveness. It maintains high task completion rates even in challenging environments like MiniGrid's ColoredDoorKey or the visually realistic Habitat environment. Overall, When2Ask successfully demonstrates the potential of combining reinforcement learning with pre-trained language models to enable cost-effective and intelligent decision-making in embodied agents.

## 2.2 Large Language Model as a Policy Teacher for Training Reinforcement Learning Agents

The paper titled "Large Language Model as a Policy Teacher for Training Reinforcement Learning Agents" introduces LLM4Teach, a framework that uses large language models (LLMs) as teachers to train lightweight reinforcement learning (RL) student agents for sequential decision-making tasks.

Key Contributions:

1. Framework Overview: The approach uses an LLM to offer high-level policy advice to a student RL agent during training. By leveraging the LLM's knowledge, the student agent can quickly learn task-specific skills.
2. Uncertainty-Aware Instructions: The LLM provides "soft" probabilistic instructions, allowing the student agent to better understand the uncertainty of various options and improve learning efficiency.
3. Dual-Loss Learning: The student agent optimizes two objectives simultaneously: aligning with the teacher's policy through a distillation loss and improving its own performance using an RL loss. The influence of the teacher gradually decreases to allow the student to learn independently.
4. Environment Testing: Experiments on MiniGrid and Habitat environments demonstrate that LLM4Teach improves the sample efficiency of RL, while also enabling the student agent to surpass the performance of its LLM-based teacher.

Results:

LLM4Teach outperformed other RL methods in sample efficiency, achieving higher returns and task success rates in complex environments like ColoredDoorKey and Nav & Pick. The framework allows the student agent to correct suboptimal or erroneous policies suggested by the LLM, ultimately achieving better performance than relying solely on LLM guidance. Overall, LLM4Teach leverages the reasoning

capabilities of LLMs to train specialized RL agents that can effectively complete target tasks with reduced training costs and computational requirements.

### 2.3 OPtions as REsponses: Grounding Behavioural Hierarchies in Multi-Agent Reinforcement Learning

The paper "OPtions as REsponses: Grounding Behavioural Hierarchies in Multi-Agent Reinforcement Learning" introduces a hierarchical agent architecture for multi-agent reinforcement learning (RL). The authors focus on enhancing agents' generalization in games by grounding strategic decision-making in the game-theoretic structure.

Key Contributions:

1. Hierarchical Agent Architecture: The proposed agent, OPtions as REsponses (OPRE), features a two-level hierarchical structure. The top level strategically chooses responses to other agents' behaviors, while the low level converts these responses into primitive actions.
2. Grounded in Game Theory: The high-level choices are based on the game's concealed information (like the opponent's strategy), while the low level implements those choices as options that guide the agent's policy.
3. Games with Non-Transitive Rewards: The authors introduce two new grid-world games (Running With Scissors and RPS Arena) with non-transitive reward structures and concealed information to test generalization. These games mimic the "rock-paper-scissors" mechanics, emphasizing strategic responses.
4. Superior Generalization: OPRE shows significantly better generalization against unseen opponents compared to other deep RL methods, which often fail to explore the strategy space thoroughly and overfit to known opponents.

Results:

The OPRE agent outperforms baseline architectures in games like Running With Scissors and RPS Arena, demonstrating superior generalization by adapting to novel opponents' strategies. OPRE learns more efficient behavioral hierarchies that exploit strategic knowledge, allowing it to maintain a high level of performance against previously unseen adversaries. Overall, OPRE showcases a method for efficiently building behavioral hierarchies that enable agents to generalize across diverse competitive environments, balancing exploration and exploitation.

### 2.4 IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures

The paper titled "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures" presents a new distributed reinforcement learning (RL) agent architecture called IMPALA.

Key Contributions:

1. Architecture Overview: IMPALA is designed with an Importance Weighted Actor-Learner Architecture, enabling efficient use of computational resources both on single and multiple machines. Actors gather trajectories of experience, while a learner asynchronously processes batches of data for policy updates.
2. V-Trace Off-Policy Correction: To handle potential policy lag between actors and the learner, the authors introduce a novel off-policy correction method, V-trace. This algorithm ensures stable learning by correcting the discrepancies between policies, making the architecture robust to large-scale distributed training.
3. High Throughput: By decoupling acting from learning, IMPALA achieves throughput rates of up to 250,000 frames per second and trains 30 times faster than single-machine A3C (Asynchronous Advantage Actor-Critic).
4. Multi-Task Learning: The architecture is evaluated on DMLab-30 (a set of 30 diverse tasks) and Atari-57 (all Atari games), demonstrating superior performance. IMPALA improves upon baseline methods in both data efficiency and stability.

Results:

IMPALA achieves better performance than A3C across various tasks, learning faster while using less data. The architecture also facilitates multi-task learning, where a single agent solves all 57 Atari games, showcasing positive transfer between tasks. Overall, IMPALA provides a scalable, efficient, and effective solution for distributed reinforcement learning, enabling significant improvements in multi-task performance.

## 2.5 Large Language Model based Multi-Agents: A Survey of Progress and Challenges

1. Background and Motivation: LLMs have shown notable progress in reasoning and planning, which makes them suitable as autonomous agents. Using multiple LLM-based agents collaboratively leads to more effective problem-solving and world simulation.
2. Architecture of LLM-MA Systems: The survey analyzes the architecture, focusing on: Interface between agents and the environment. Profiling methods that describe agent characteristics. Communication paradigms and structures used among agents. Capability Acquisition strategies that enhance agent skills.
3. Applications: LLM-based multi-agent systems have been applied to a range of domains: Problem-Solving: Software development, robotics, science experiments, and medical diagnosis. World Simulation: Social simulation, gaming, economics, psychology, and policy-making.
4. Challenges and Opportunities: Key challenges include advancing to multi-modal environments, reducing hallucination, improving collective intelligence, and scaling up the systems.
5. Resources: The survey also lists datasets, benchmarks, and open-source frameworks like MetaGPT, CAMEL, and Autogen, which researchers can use to develop and evaluate LLM-based multi-agent systems.

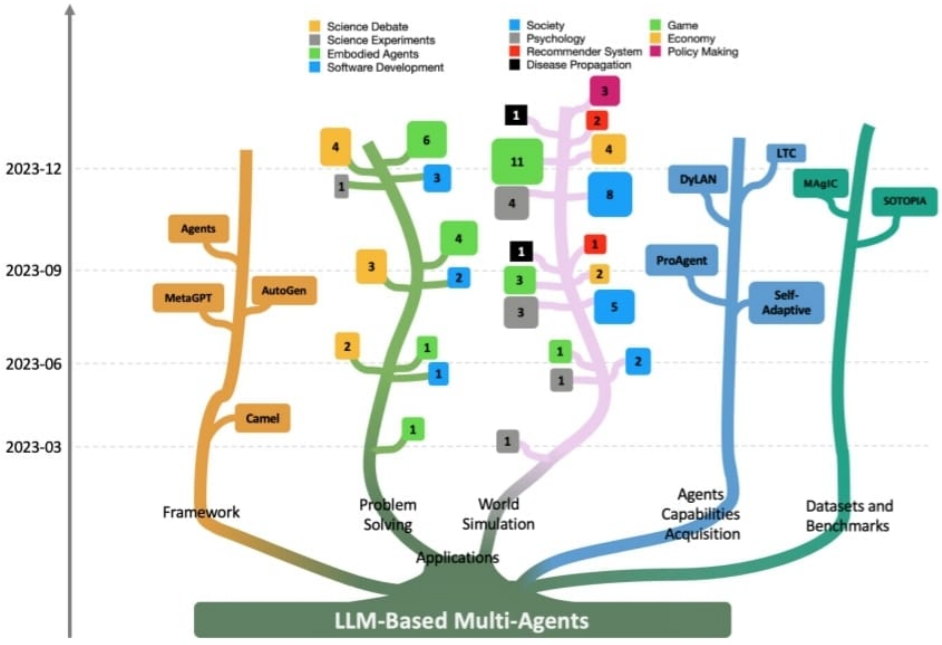



Figure 1: The rising trend in the research field of LLM-based Multi-Agents. For Problem Solving and World Simulation, we categorize current work into several categories and count the number of papers of different types at 3-month intervals. The number at each leaf node denotes the count of papers within that category.

## 2.6 Decision Transformer: Reinforcement Learning via Sequence Modeling

The "Decision Transformer" paper introduces a framework that recasts reinforcement learning as a sequence modeling problem using the Transformer architecture. By treating reinforcement learning as a conditional sequence modeling task, the Decision Transformer outputs optimal actions based on desired return, past states, and previous actions. It achieves this without traditional value functions or policy gradients, relying instead on the capabilities of Transformers.

Key points:

1. Framework: Uses a GPT-like autoregressive Transformer model conditioned on desired return.
2. Evaluation: Matches or exceeds state-of-the-art offline reinforcement learning models on Atari, OpenAI Gym, and Key-to-Door tasks.
3. Benefits: Handles sparse rewards well, performs effective long-term credit assignment, and doesn't require behavior regularization or value pessimism.

This approach leverages the scalability of Transformers, simplifying reinforcement learning while maintaining strong performance across tasks.

# 3 MARL-JAX

## 3.1 Abstract

Recent advances in Reinforcement Learning (RL) have led to many exciting applications. These advancements have been driven by improvements in both algorithms and engineering, which have resulted in faster training of RL agents. We present marl-jax, a multi-agent reinforcement learning software package for training and evaluating social generalization of the agents. The package is designed for training a population of agents in multi-agent environments and evaluating their ability to generalize to diverse background agents. It is built on top of DeepMind's JAX ecosystem Babuschkin et al. (2020) and leverages the RL ecosystem developed by DeepMind. Our framework marl-jax is capable of working in cooperative and competitive, simultaneous-acting environments with multiple agents. The package offers an intuitive and user-friendly command-line interface for training a population and evaluating its generalization capabilities. In conclusion, marl-jax provides a valuable resource for researchers interested in exploring social generalization in the context of MARL

## 3.2 Architecture

The following 4 architecture types are implemented in the paper:

1. Single Threaded RL
2. Synchronous Distributed
3. IMPALA-style Asynchronous Distributed
4. Sebulba: Asynchronous Distributed with Inference Server

### 3.2.1 Single Threaded RL

At each step, the agent receives an observation from the environment, selects an action based on its policy, interacts with the environment, and receives a reward. The observation, action, reward, and next observation are collected to form a batch and then used to update agent's policy and value function using gradient descent. This process continues iteratively until the desired convergence or a specified number of iterations is reached.

### 3.2.2 Synchronous Distributed

The synchronous distributed architecture builds upon the single-threaded architecture by utilizing multiple environment instances running in parallel processes to collect a batch of experiences simultaneously. Each environment synchronously interacts with a common policy, generating sequences of states, actions, rewards, and next observations. This batch of sequences is used to update the policy and value function weights through gradient descent. By leveraging parallelization, the synchronous distributed architecture

enables more efficient data collection and faster updates, leading to accelerated training and improved convergence in reinforcement learning.

#### 3.2.3 IMPALA-style Asynchronous Distributed

In the IMPALA-style asynchronous distributed training architecture, multiple actors run in parallel and interact with their respective environments asynchronously. Each actor collects trajectories of experience by executing its own copy of the current policy. The three main components running in parallel as separate processes are described below

1. Environment Loop: The environment loop process interacts with the environment using the available policy and adds the collected experience to the replay buffer. Multiple parallel environment loop processes are run, each with its own copy of the environment and policy parameters. We use CPU inference for action selection on each process. To keep the policy parameters in sync with the learner process, the parameters are periodically fetched from the learner process. The action selection step is optimized using vamp auto-vectorization to select the action for all agents in the environment.
2. Learner: The actual policy learning happens in this process. The learner fetches experience from the replay buffer and performs the optimization step on policy and value function parameters. We use pmap to auto-scale the optimization step to multiple GPUs and vmap based auto-vectorization to perform the optimization step for all agents in parallel. Algorithm 3 shows the pseudocode for this process.
3. Replay Buffer: A separate process with reverb Cassirer et al. (2021) server is used as a replay buffer. All the actors add experience to this server, and the learner process samples experience from the server to optimize for policy and value function parameters.

# 4 MARL JAX Code Structure

## 4.1 ACME

Developed for single agent reinforcement learning research and is the major library used in MARL-JAX. Ideally MARL-JAX is just using multiple agents instead of acme as single agent with similar code adaptation.

## 4.2 rLax

This library deals with reinforcement learning algorithms abstracted as a single function. It provides mathematical tools for calculating entropy , policy gradient loss , vtrace , etc.

## 4.3 Haiku

This library is efficiently used in the feature extraction and neural network based models running at back of each agent. This is basically built on top of JAX and can be thought of as a NN library.

## 4.4 Reverb

This is a library used to store the (Observation, Action, Reward) tuples in the experience replay buffer tables. It gives custom abilities to sample the experience replays for training.

## 4.5 Optax

This library is used for just gradient descent and optimizers. As such it has no extra use in the codebase.Basically, for training the LSTM network they are using its optimizer.

## 4.6 dm_env

This is the library that provides the deepmind environments wrapper APIs for running our RL algorithms in it. It major use is to retrieve the (Observation, Action, Reward) tuples from the environment in simulation.

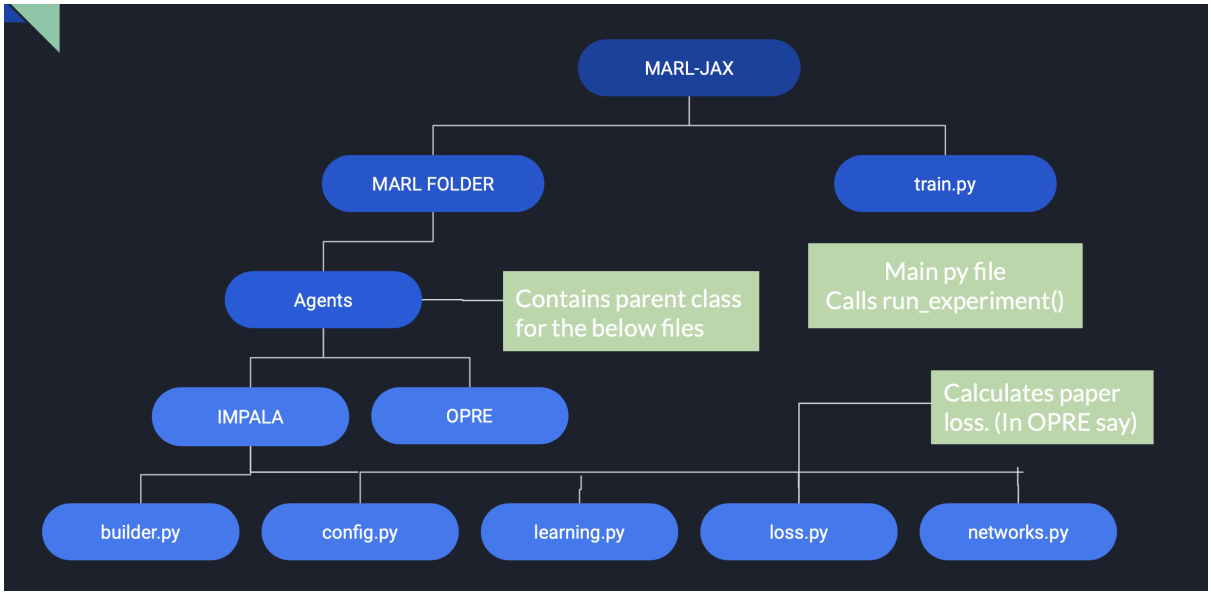



Figure 1: High Level Overview of the Codebase

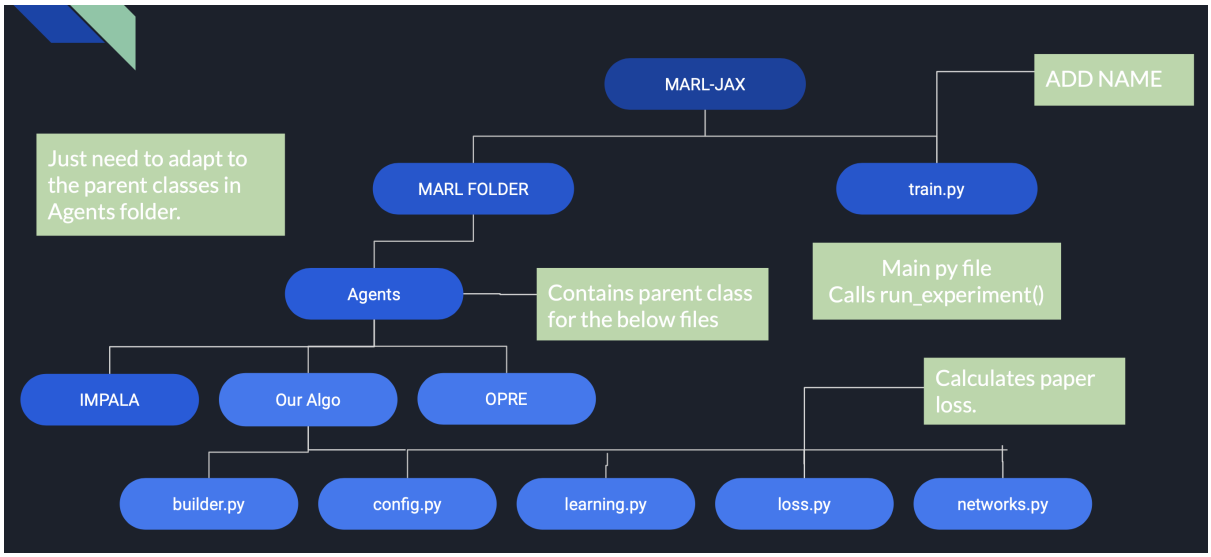



Figure 2: Proposed Additions in the Codebase

# 5 Prompt

LLM Prompt for Overcooked

prompt="Hello! In this task, you will help the two agents play a cooking game. Both of the agents (chefs) have an aim to serve the onion soup. The game map consists of the dishes for serving , the ingredients to make the soup like the onions , the pots to cook the soup and the serving counter to finally deliver the soup. Take the map as a grid, where moving in x direction means moving towards EAST amd moving in y direction measn moving towards WEST. There are two agents in the map that cooperate among themselves to deliver as many soups as possible while minimizing the time. Your task will be to give an output of optimal actionas for both agents in a list : [a1,a2] based on the given observations BELOW. The observations contain agent 1's and agent 2's observations. Description for the actions (a) : 1) a = 0 means to move one step towards NORTH. 2) a = 1 means to move one step towards SOUTH. 3) a = 2 means move one step towards EAST. 4) a = 3 means move one step towards WEST. 5) a = 4 means STAY at your position. Don't move. 6) a = 5 means INTERACT. So you can use this action to pick up different objects and items in the map as soon as they are (0,0) distance apart (both in x and y direction).
Observation for agent 0 : The player 0 is facing towards the NORTH direction. He is holding nothing in his hand along with him. As told that he is in a gridworld, here are his distances (in (dx,dy) such that dx = x-coordinate of item - x-coordinate of player 0 and dy = y-coordinate of item - y-coordinate of player 0) : onion is [-1 -1] distance apart , tomato is [0 0] distance apart , dish is [0 1] distance apart , soup is [0 0] distance apart , closest serving is [2 1] distance apart , empty counter is [0 0] distance apart , . There are 0 onions and 0 tomatoes in the closest soup. The Pot numbers in the list [0] exist and are visible, the status of the respective pots are also given in this list : ['is empty'], they have the [0] number of onions and [0] number of tomatoes (listed per pot). The cooking time of these available pots are also listed : [0] (in minutes). Please note that if no soup is cooking in a particular pot, cook time is -1. The distance of these pots from the agent are also described in a similar fashion as the above described (dx,dy) coordinates , here is the list for the respective pots : [[1, -2]]. The player has walls in his SOUTH,WEST direction(s). Player 0 is at a distance of 2 along the x-direction and -1 along the y-direction.Player 0 is at the position coordinate [1 2] in the grid.
Observation for agent 1 : The player 1 is facing towards the NORTH direction. He is holding nothing in his hand along with him. As told that he is in a gridworld, here are his distances (in (dx,dy) such that dx = x-coordinate of item - x-coordinate of player 1 and dy = y-coordinate of item - y-coordinate of player 1) : onion is [1 0] distance apart , tomato is [0 0] distance apart , dish is [-2 2] distance apart , soup is [0 0] distance apart , closest serving is [0 2] distance apart , empty counter is [0 0] distance apart , . There are 0 onions and 0 tomatoes in the closest soup. The Pot numbers in the list [0] exist and are visible, the status of the respective pots are also given in this list : ['is empty'], they have the [0] number of onions and [0] number of tomatoes (listed per pot). The cooking time of these available pots are also listed : [0] (in minutes). Please note that if no soup is cooking in a particular pot, cook time is -1. The distance of these pots from the agent are also described in a similar fashion as the above described (dx,dy) coordinates , here is the list for the respective pots : [[-1, -1]]. The player has walls in his NORTH,EAST direction(s). Player 1 is at a distance of -2 along the x-direction and 1 along the y-direction.Player 1 is at the position coordinate [3 1] in the grid."

This prompt was specifically designed for the Overcooked game, a multi-agent reinforcement learning (MARL) environment where agents (players) must cooperate to achieve the objective of serving as many onion soups as possible. To enable large language models (LLMs) to provide strategic suggestions for this game, the prompt was created with detailed information about the locations of agents and game items.

How the Prompt Was Created:

1. Parsing Locations and Observations: The positions and distances of both agents and various objects in the game (e.g., onions, tomatoes, dishes, soup, serving counters) were parsed through code and translated into coordinates. This information was processed for each agent separately to reflect their unique perspectives.

2. Environment Description: The prompt begins by explaining the overall game objective and the role of the two agents (chefs). It then describes the map's grid structure and the possible actions each agent can take (e.g., moving in various directions, interacting with items).
3. Individual Observations: The prompt includes separate observations for each agent, detailing:
4. Relative Coordinates: The relative coordinates of important items like ingredients, dishes, pots, and counters from each agent's location.
5. Item Status: The state of ingredients and soups (e.g., how many are present or if a pot is empty).
6. Obstacles and Position: The walls around the agent and their position on the grid map.
7. Action Explanation: Each possible action (e.g., moving in a direction or interacting with objects) is explicitly defined, making it clear how agents can navigate and manipulate the environment.

The prompt provides the LLM with comprehensive information, allowing it to understand the agents' environment fully and generate lists of optimal actions for both agents based on the parsed observations. By synthesizing this structured data, the LLM can suggest strategies that will help both agents coordinate and achieve the primary goal effectively.

# 6 Experiments

We evaluate our implementation in two environments to assess its performance and generalization capabilities across different types of multi-agent scenarios.

## 6.1 LLM4RL: Challenges in Mapping LLM Responses to Actions

In our exploration of the LLM4RL codebase, we aimed to understand how Large Language Models (LLMs) can efficiently interact with algorithmic agents in reinforcement learning (RL) tasks. Specifically, we focused on the challenge of mapping LLM responses to actions, a crucial aspect of enabling effective communication between the agent and the LLM.

Our experimentation revealed several insights regarding the mapping of LLM responses to actions within the LLM4RL framework:

1. Hard-coded Mapping: We found that the mapping of LLM responses to actions was hard-coded within the codebase. This meant that the actions taken by the agent in response to LLM instructions were predefined and not adaptable based on the specific task or environment. This hard-coded approach limited the flexibility and scalability of the system, as it did not allow for dynamic adjustments in response to changing conditions.
2. Lack of Optimization: The hard-coded mapping of LLM responses to actions was not optimized for scalability or efficiency. Since the mapping was predefined, it did not take into account the varying complexity of tasks or the potential for nuanced instructions from the LLM. As a result, the agent's performance may have been suboptimal in certain scenarios where a more nuanced response was required.
3. Limited Adaptability: The lack of adaptability in the mapping of LLM responses to actions restricted the agent's ability to learn and improve over time. Without the flexibility to adjust the mapping based on experience and feedback, the agent may struggle to effectively utilize the information provided by the LLM, particularly in dynamic or uncertain environments.

**Challenge :** Th major challenge for the codebase was to set up an open API for LLM prompting and chat completion. The GPT , Bard and Gemini APIs were tried but they were paid after a limit. Finally, we had to compromise on the model so we took a quantized VICUNA-7B chat completion fine tuned model. THis was recommended by the authors. But it was giving wrong outputs even when we gave a choice of 4 options. This made the policy framework fail and didn't do well !

Overall, our experimentation with the LLM4RL codebase highlighted the importance of developing more adaptive and scalable approaches for mapping LLM responses to actions in RL tasks. By addressing the limitations of hard-coded mappings and optimizing the mapping process for efficiency and adaptability, we can enhance the effectiveness of LLMs in assisting algorithmic agents in reinforcement learning

settings. Further research in this area is needed to explore more advanced techniques for facilitating communication and collaboration between LLMs and RL agents.

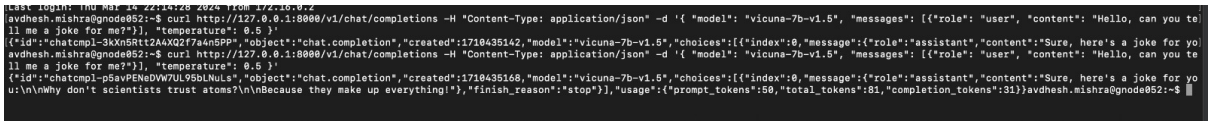



Figure 3: Vicuna-7B hosted locally

## 6.2 OvercookedGPT

In our experimentation with OvercookedGPT, we encountered some limitations that influenced our choice of Large Language Models (LLMs) for evaluation. Specifically, we were unable to test with GPT due to limited access to the API. As a result, we opted to use Anthropic's Claude for our experiments instead.

Despite the initial constraint, our use of Claude provided valuable insights into the capabilities of LLMs within the OvercookedGPT framework. We were able to assess Claude's performance in long-horizon reasoning, task planning, and coordination of multiple agents in dynamic environments.

While our original intention was to compare multiple LLMs, including GPT, the use of Claude allowed us to proceed with our evaluation and gather meaningful results. Our experience underscores the importance of flexibility and adaptability in research settings, where unforeseen constraints may necessitate adjustments to experimental plans.

Overall, our experiment with Claude in OvercookedGPT provided valuable insights into the potential applications of LLMs in multi-agent environments, despite the initial limitation of not being able to test with GPT directly.

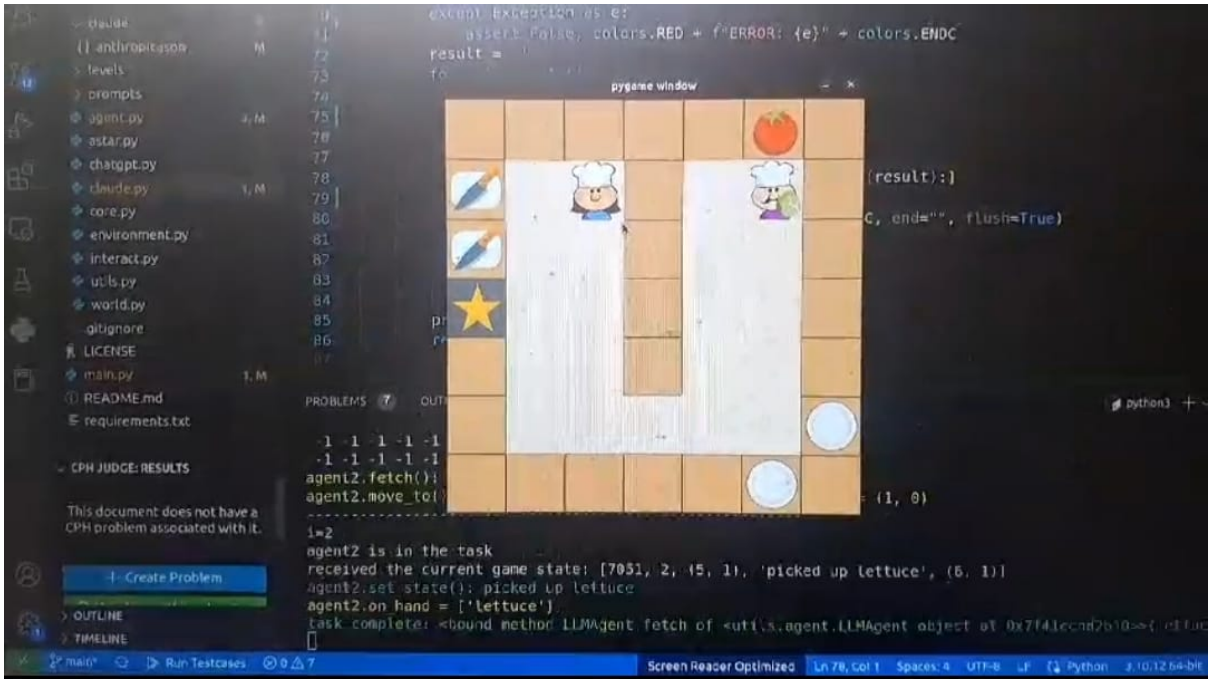



Figure 4: Overcoooked GPT running on the LLM

# 7 Code

## 7.1 MARL-JAX

### 7.1.1 Observation Parsing

This function returns the observation encoding vector for each agent in the environment. It is useful in interpreting the observations as rule based designs. This helps to easily parse it using a python script and convert to a chat prompt for LLMs.

```python
def _get_observation(self) -> list[types.NestedArray]:
    # observation = self.env.lossless_state_encoding_mdp(self.env.state)
    observation= self.env.featurize_state_mdp(self.env.state)
    return [{"agent_obs": obs} for obs in observation]
```

Figure 5: Parsing Agent Observations

### 7.1.2 Building prompts through Agent Observations

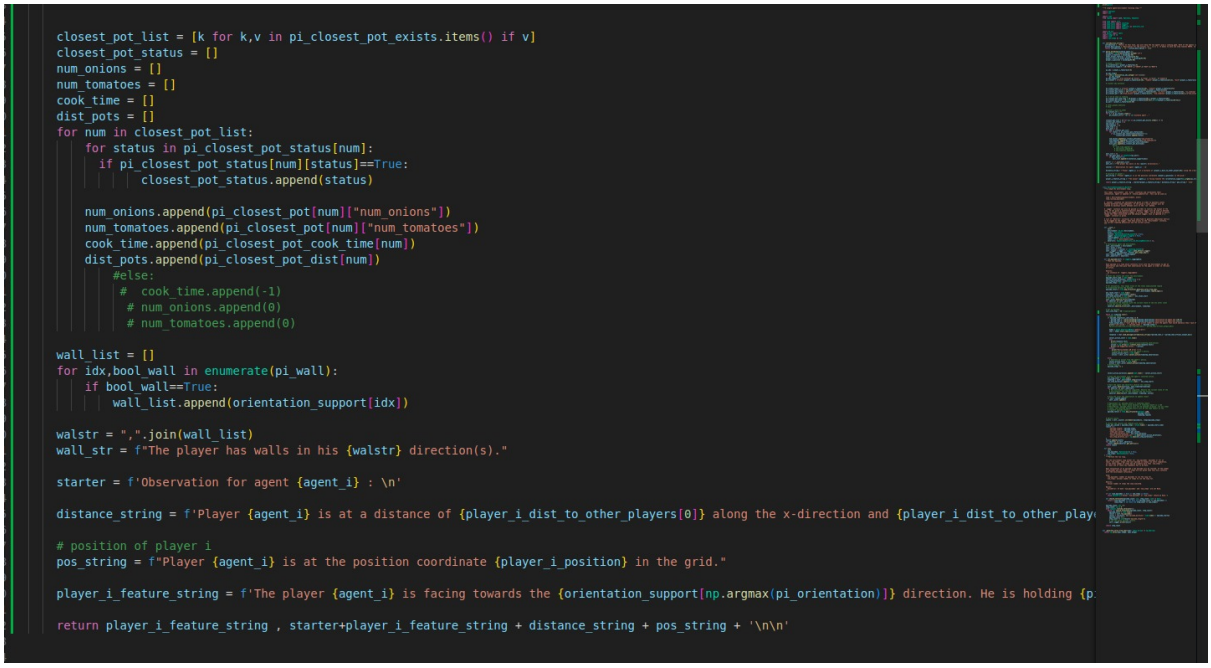
This code generates prompts for a multi-agent reinforcement learning environment by analyzing the observations of each agent. The introduction_string function provides a game introduction and action descriptions.

The parse_encoding function processes encoded data to capture each agent's unique observations, including their orientation, nearby objects (e.g., onions, tomatoes), pot status, and obstacles like walls. These details are formatted into strings that describe the environment from the agent's perspective.

The prompts created by combining these strings allow a large language model to understand the multi-agent environment and offer strategic suggestions.

```python
def introduction_string():
    introduction =  "Hello! In this task, you will help the two agents play a cooking game. Both of the agents (chefs) have an aim to serve the onion so
    action_description = "Description for the actions (a) : \n 1) a = 0 means to move one step towards NORTH. \n 2) a = 1 means to move one step towards
    return introduction + '\n' + action_description +' \n\n'

def parse_encoding(encoding,agent_i):
    encoding = np.array(encoding).astype('int')
    player_i_features = encoding[:46]
    other_player_features = encoding[46:92]
    player_i_dist_to_other_players = encoding[92:94]
    player_i_position = encoding[94:96]

    # player_i_features
    pi_orientation = player_i_features[:4]
    orientation_support = {0:"NORTH",1:"SOUTH",2:"EAST",3:"WEST"}

    pi_obj = player_i_features[4:8]

    pi_obj_flag=1
    if (len(np.bincount(pi_obj.astype('int')))==1):
        pi_obj_flag=0
    pi_obj_support = {-1:"nothing",0:"onion", 1:"soup", 2:"dish", 3:"tomato"}
    pi_closest = {"onion":player_i_features[8:10], "tomato":player_i_features[10:12], "dish":player_i_features[12:14], "soup":player_i_features[14:16],

    # closest obj distance


    pi_cloest_soup_n = {"onion":player_i_features[16], "tomato":player_i_features[17]}
    pi_closest_pot_exists = {0:player_i_features[22],1:player_i_features[32]}
    pi_closest_pot_status = {0:{"is_empty":player_i_features[23], "is_full":player_i_features[24], "is_cooking" : player_i_features[25], "is_ready":pla
    pi_closest_pot = {0:{"num_onions":player_i_features[27], "num_tomatoes":player_i_features[28]},1:{"num_onions":player_i_features[37], "num_tomatoes

    # -1 if no soup is cooking
    pi_closest_pot_cook_time = {0:player_i_features[29],1:player_i_features[39]}
    pi_closest_pot_dist = {0:list(player_i_features[30:32]),1:list(player_i_features[40:42])}
    pi_wall = player_i_features[42:46]

    # other_player_features
    # done

    # player_i_dist_to_other
    pi_closest_str = ""
    for k,v in pi_closest.items():
        pi_closest_str+=f" {k} is {v} distance apart , "
```

Figure 6: Parsing Agent Observations



```
closest_pot_list = [k for k,v in pi_closest_pot_exists.items() if v]
closest_pot_status = []
num_onions = []
num_tomatoes = []
cook_time = []
dist_pots = []
for num in closest_pot_list:
    for status in pi_closest_pot_status[num]:
      if pi_closest_pot_status[num][status]==True:
            closest_pot_status.append(status)

    num_onions.append(pi_closest_pot[num]["num_onions"])
    num_tomatoes.append(pi_closest_pot[num]["num_tomatoes"])
    cook_time.append(pi_closest_pot_cook_time[num])
    dist_pots.append(pi_closest_pot_dist[num])
        #else:
         #  cook_time.append(-1)
          # num_onions.append(0)
          # num_tomatoes.append(0)

wall_list = []
for idx,bool_wall in enumerate(pi_wall):
    if bool_wall==True:
        wall_list.append(orientation_support[idx])

walstr = ",".join(wall_list)
wall_str = f"The player has walls in his {walstr} direction(s)."

starter = f'Observation for agent {agent_i} : \n'

distance_string = f'Player {agent_i} is at a distance of {player_i_dist_to_other_players[0]} along the x-direction and {player_i_dist_to_other_play

# position of player i
pos_string = f"Player {agent_i} is at the position coordinate {player_i_position} in the grid."

player_i_feature_string = f'The player {agent_i} is facing towards the {orientation_support[np.argmax(pi_orientation)]} direction. He is holding {p

return player_i_feature_string , starter+player_i_feature_string + distance_string + pos_string + '\n\n'
```

Figure 7: The functionality to parse the observation encoding to convert it into chat prompt and call for LLM response through LLM API was integrated in MARL-JAX.

### 7.1.3 LLM calling

Once we have the final prompt to be passed to the LLM, this code integrates a large language model (LLM) into a multi-agent reinforcement learning (MARL) environment, specifically calling the LLM every 50 actor steps while training to provide strategic instructions that can enhance generalization.

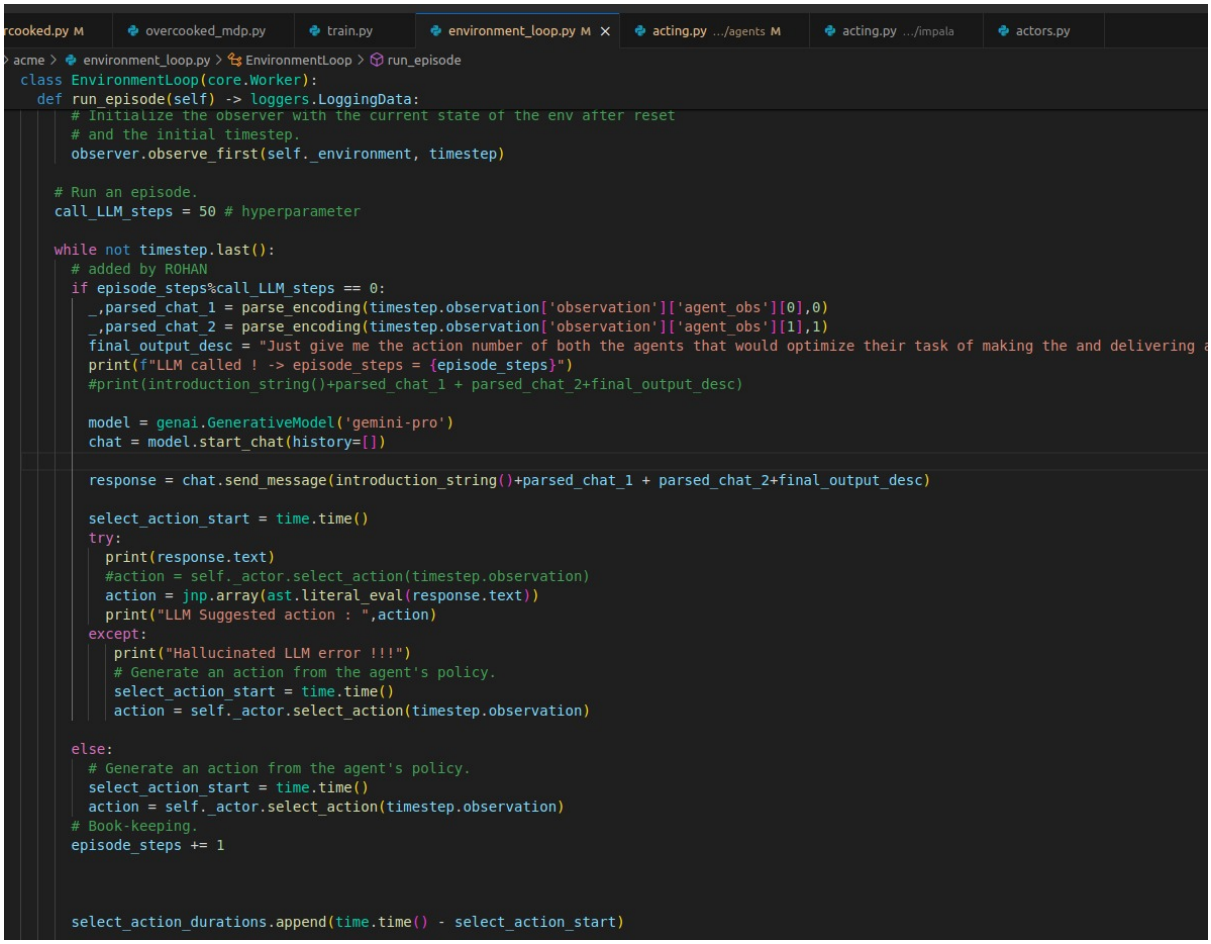



Figure 8: Parsing Agent Observations

# 8 Outputs

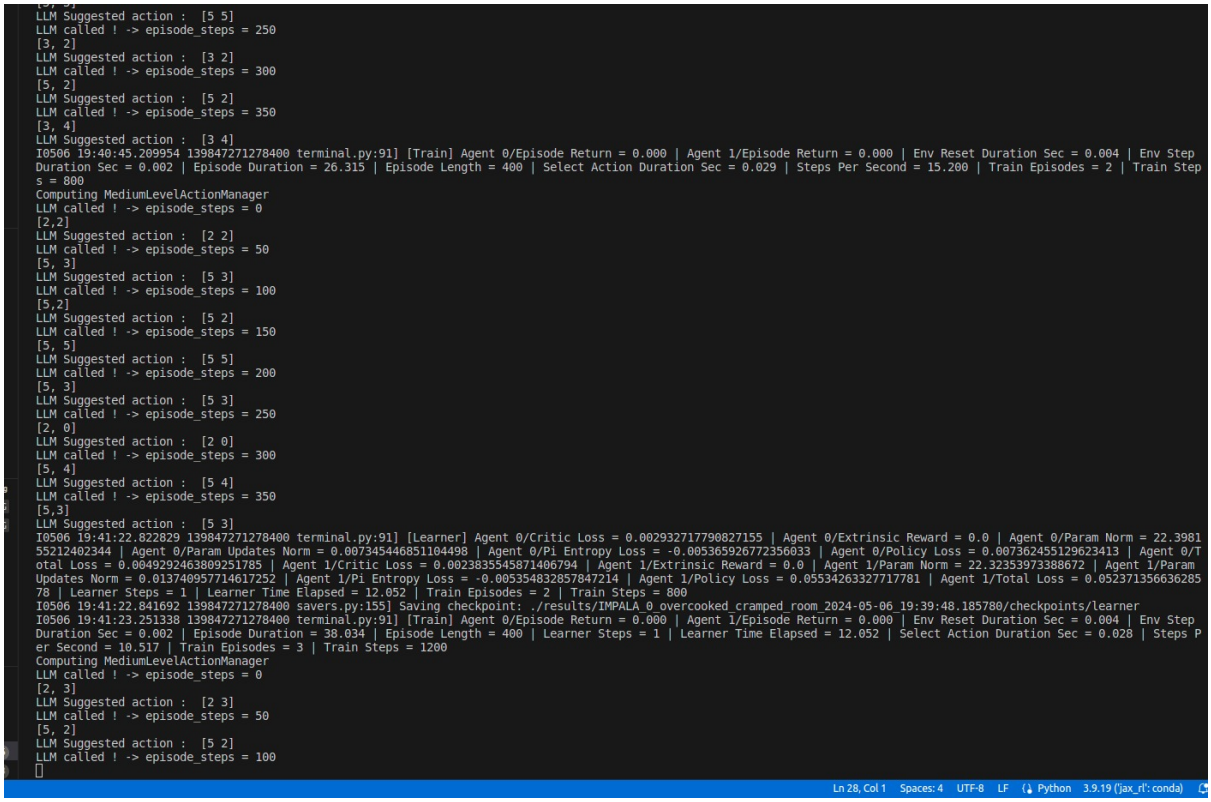



Figure 9: LLM suggests actions every 50 actor steps

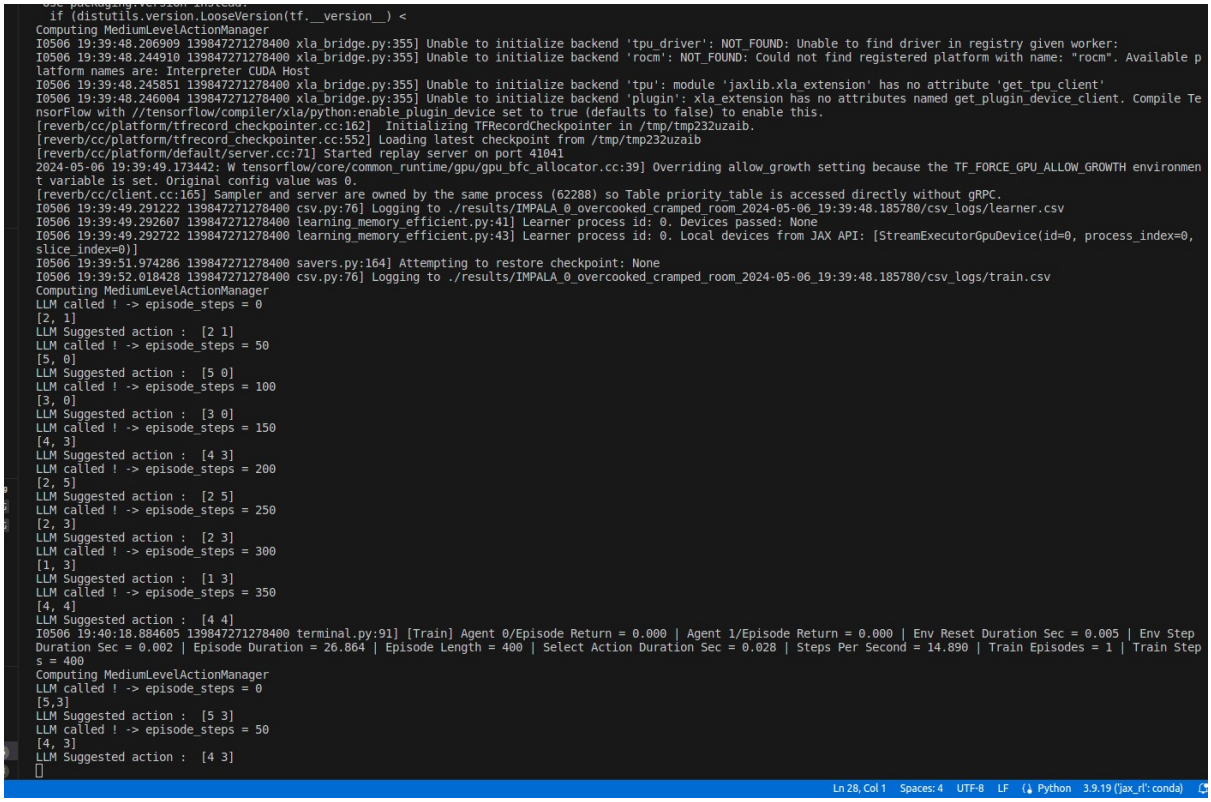



Figure 10: LLM suggests actions every 50 actor steps

# 9 Final Results

## 9.1 Without LLM integration

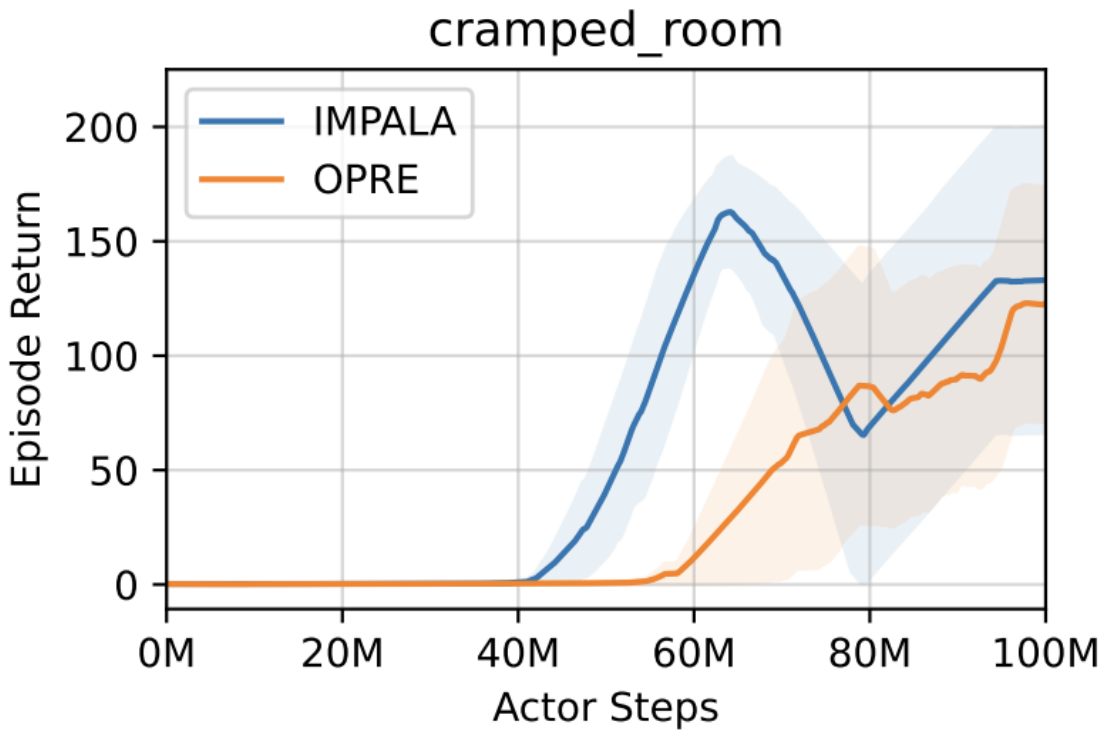



Figure 11: Training Plot on Cramped Room

## 9.2 With LLM integration

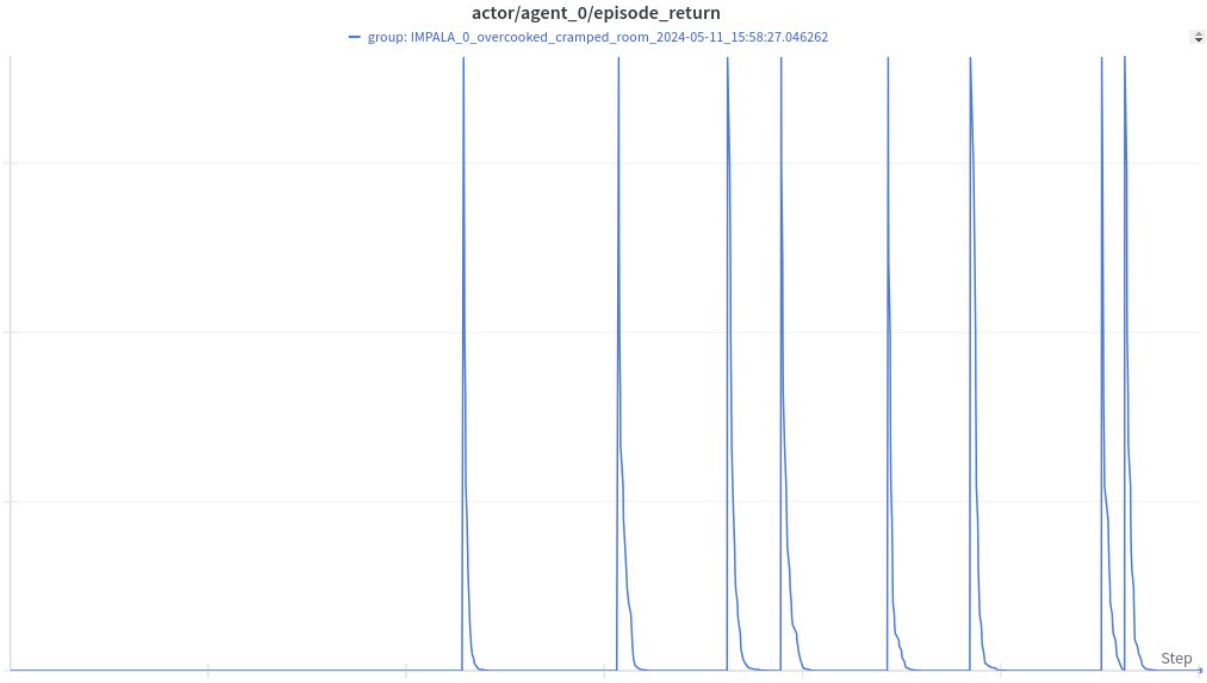



Figure 12: Training Plot on Cramped Room with LLM integration

### 9.3 Average episode return

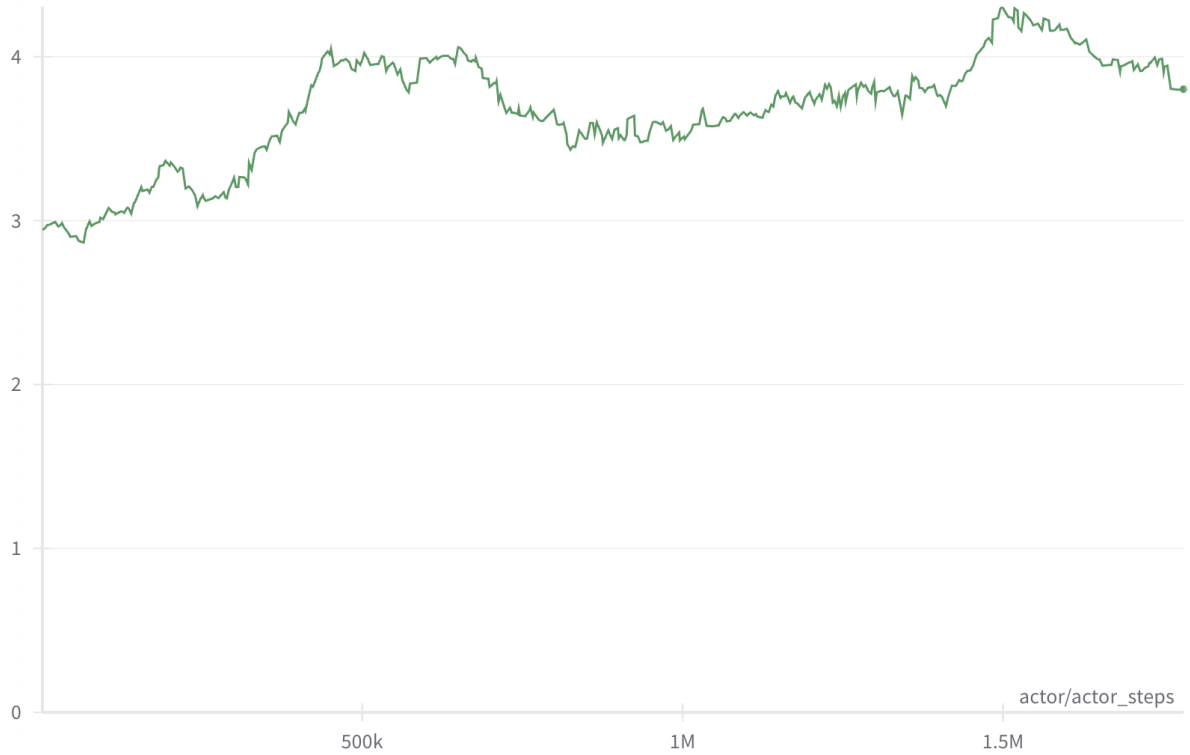



Figure 13: Training Plot on Cramped Room with LLM integration

In the conducted experiments, the agent achieved a significant level of performance within 4 million actor steps, successfully returning positive episodic returns. This is in stark contrast to the baseline where, in the absence of language model guidance, no positive returns were observed even after an extended period of 40 million steps. These results suggest that the integration of large language models (LLMs) into the IMPALA framework markedly enhances its efficacy. The observed increase in the magnitude of episodic returns further supports the conclusion that the LLMs effectively contribute valuable feedback, optimizing the learning process within the IMPALA architecture.