h e g

Haute école de gestion
Genève

Bridging Reasoning and Reactive Execution in Multi-Agent Reinforcement Learning

Master's thesis by:

Dorian THOMÉ

Under the direction of:

Alexandros KALOUSIS, Prof. Dr.

Geneva, August 15, 2026

Information Sciences
Haute École de Gestion de Genève (HEG-GE)

Hes·so GENÈVE
Haute Ecole Spécialisée de Suisse occidentale

Declaration

This Master's thesis is carried out as part of the final school examination, with a view to obtaining the title Master of Science HES-SO in Information Sciences.

The student attests that he has submitted the work to a plagiarism detection software. The student accepts the confidentiality clause, if applicable. The use of the conclusions and recommendations formulated in the Master's thesis, without prejudice to their value, does not engage the responsibility of the author, the Master's thesis, advisor, the jury or the HEG-GE.

In accordance with HEG-GE guidelines, the author declares the use of Generative AI tools (such as Google Gemini) exclusively for proofreading, stylistic refinement, coding assistance and LaTeX formatting assistance. All intellectual content, experimental data, and conceptual architecture remain the sole original work of the author.

"I hereby certify that this work has been carried out using only the sources cited in the bibliography, that it is the fruit of my personal reflection and has been written independently."

Done at Geneva, 15th of August 2026

Dorian THOMÉ

Acknowledgments

My gratitude to my supervisor, Prof. Dr. Alexandros Kalousis, and Dr. Jacopo Castellini for their guidance and support throughout this research.

Summary

Reinforcement learning agents struggle in environments with sparse rewards. This thesis investigates combining Large Language Models (LLMs) with Multi-Agent Reinforcement Learning (MARL) to resolve exploration bottlenecks. The primary objective is to evaluate whether the semantic reasoning of an LLM can guide MAPPO agents through interdependent tasks modeled as a Directed Acyclic Graph (DAG).

We developed and evaluated two distinct neuro-symbolic architectures. The first approach, Dynamic Reward Shaping, utilizes the LLM as a strategic supervisor that continuously adjusts the priority weights of a Potential-Based Reward Shaping (PBRS) signal. The second approach introduces a Hierarchical Reinforcement Learning (HRL) Meta-Controller, where the LLM assigns discrete macro-actions (Options) to the agents, directly modifying their observation space and intrinsic rewards. Both architectures were evaluated through ablation studies targeting temporal smoothing, zero-shot semantic adaptation, QLoRA fine-tuning, and counterfactual reflection.

The results demonstrate that the HRL architecture significantly outperforms dynamic shaping, enabling agents to master the entire environment, including complex combat coordination. However, integrating semantic models with kinetic policies introduces significant non-stationarity. Implementing Exponential Moving Average (EMA) smoothing for the Dynamic architecture and Kullback-Leibler (KL) divergence guardrails for the HRL architecture proved necessary to maintain policy stability.

Furthermore, the research highlights two fundamental vulnerabilities in neuro-symbolic systems. First, the LLM's semantic overconfidence can induce reward hacking, trapping agents in local optima. Second, counterfactual reflection triggers over-correction to simple physical delays, actively degrading learning efficiency. Prospectively, while QLoRA fine-tuning accelerated convergence five-fold, the viability of these architectures for high-frequency control will require the development of foundation models equipped with spatio-temporal memory.

Keywords: Multi-Agent Reinforcement Learning (MARL), Large Language Models (LLMs), Hierarchical Reinforcement Learning (HRL), Reward Shaping, Proximal Policy Optimization (PPO), Neuro-Symbolic AI, QLoRA.

Table of contents

List of Tables

List of Figures

1 Introduction

1.1 Context: Multi-Agent Systems in Open-World Environments

Multi-Agent Reinforcement Learning (MARL) provides a mathematical framework for training decentralized agents to maximize a shared or competitive utility. As the field has matured, researchers have increasingly utilized Massively Multiplayer Online Role-Playing Games (MMORPGs) and open-world environments, such as Neural MMO and Minecraft, as testbeds for real-world complexity. These environments require agents to navigate vast state spaces, manage resources, and engage in high-level strategic coordination.

To solve these tasks, the Centralized Training with Decentralized Execution (CTDE) paradigm has emerged as the standard architecture. Under this paradigm, algorithms such as Multi-Agent Proximal Policy Optimization (MAPPO) (Yu et al. 2022) have shown strong results in cooperative settings. By utilizing a centralized critic that has access to the global state during training, agents can learn robust decentralized execution policies that rely solely on local observations.

1.2 The Exploration Bottleneck in Sparse-Reward Worlds

Despite the success of algorithms like MAPPO, pure deep reinforcement learning struggles catastrophically in environments characterized by long-horizon, sparse-reward structures. In complex RPG-style environments, progress is often gated behind strict dependency graphs (e.g., agents must jointly collect wood and stone, navigate to a workbench, craft a pickaxe, and then coordinate to mine iron).

Traditional RL agents rely on random exploration to discover these initial reward signals. However, as the depth of the dependency chain increases, the probability of agents randomly executing the exact sequence of required motor commands approaches zero. While engineers traditionally attempt to solve this via dense, hand-crafted reward shaping, this introduces significant risks. Poorly designed shaping signals frequently lead to reward hacking, where agents optimize for surrogate metrics without achieving the true environmental milestone (Ma et al. 2024). At the same time, as multiple agents update their policies simultaneously, the environment becomes highly non-stationary from the perspective of any individual agent, leading to variance spikes and policy collapse (Lowe et al. 2017). The fundamental problem lies in bridging the gap between high-level semantic planning (knowing *what* to do) and low-level kinetic control (learning *how* to do it) without destabilizing the multi-agent learning process.

To systematically investigate this intersection of large language models and multi-agent reinforcement learning, this thesis poses the following three central research questions:

RQ1: How can a high-level, semantic language model be effectively integrated with high-frequency, low-level MARL agents to solve long-horizon tasks without destabilizing the training loop?

RQ2: Between continuous dynamic reward shaping and discrete Hierarchical Reinforcement Learning (HRL), which architecture provides better sample efficiency and agent coordination?

RQ3: What practical failure modes arise when coupling LLMs with MARL (such as semantic

over-confidence or reactive self-debugging), and how can mathematical guardrails mitigate these risks?

1.3 Proposed Approach: Neuro-Symbolic MARL

To resolve this exploration bottleneck, this thesis proposes and evaluates two neuro-symbolic MARL architectures. We treat the learning process as a dual-tier problem, introducing Large Language Models (LLMs) as high-level semantic reasoners coupled with low-level MAPPO execution policies (Yu et al. 2022). This approach leverages the pre-trained world knowledge of LLMs to synergize reasoning and acting (Yao et al. 2023), bypassing the need for the agents to discover basic logical dependencies via trial-and-error.

Specifically, the subject is addressed through two implementations. The first, termed the Mixing Board, relies on dynamic reward shaping. The LLM acts as a meta-supervisor that observes aggregate training statistics and dynamically adjusts priority weights over subtasks. These weights govern a continuous Potential-Based Reward Shaping (PBRS) signal (Ng, Harada et Russell 1999), which guides the MAPPO agents toward critical bottlenecks while theoretically preserving the optimal policy.

The second implementation, the Meta-Controller, transitions to a discrete hierarchical reinforcement learning architecture. Building upon the Options framework (Sutton, Precup et Singh 1999) and hierarchical meta-controllers (Kulkarni et al. 2016), the LLM replaces the traditional symbolic planner by issuing temporally extended macro-actions directly to the agents. The agents' observation space is augmented with one-hot option encodings, and strict Kullback-Leibler (KL) divergence guardrails enforce trust region constraints during policy updates. This approach allows the agents to master the complete dependency graph and adapt to zero-shot semantic shifts in the environment. Through both architectures, this work demonstrates how the zero-shot planning capabilities of quantized, self-hosted language models (QLoRA) (Dettmers et al. 2023) can be integrated into deep reinforcement learning pipelines to solve complex cooperative tasks.

1.4 Constraints and Sustainability

A critical constraint, and a core design philosophy, of this thesis is the reliance on localized, consumer-grade computing. The contemporary trend in neuro-symbolic AI heavily favors routing reasoning tasks to massive, proprietary models via commercial APIs. However, this paradigm introduces severe latency, raises privacy concerns, and incurs prohibitive financial costs when queried across millions of MARL training steps. Furthermore, it relies on foundation models with significant training energy costs (Strubell, Ganesh et McCallum 2019).

To address these practical limitations, this work is explicitly constrained to operate entirely on localized hardware. By utilizing heavily quantized, self-hosted language models optimized via QLoRA, we demonstrate that complex, hierarchical multi-agent coordination can be achieved without cluster-scale computing. This commitment to "Green AI" (Schwartz et al. 2020) not only minimizes the energetic footprint of the training loop, but also proves that such architectures are viable for future deployment on resource-constrained edge devices, such as robotic swarms or autonomous systems operating on strict power budgets.

1.5 Thesis Outline

The subsequent chapters establish the theoretical foundations of Multi-Agent Reinforcement Learning and Large Language Models before detailing the experimental testbed. To rigorously evaluate the proposed architectures, the environment is structured as a strict task dependency graph, specifically, a Directed Acyclic Graph (DAG) where higher-level goals are gated by sequential milestones. Against this backdrop, the thesis presents its core contributions: the continuous dynamic reward shaping framework (Mixing Board) and the discrete Hierarchical Reinforcement Learning architecture (Meta-Controller). Following the architectural design, the experimental results are analyzed to compare sample efficiencies and isolate the critical failure modes of neuro-symbolic coupling. Finally, the work concludes with a discussion on future directions for sustainable, edge-viable MARL systems.

2 Background: Multi-Agent Reinforcement Learning

This chapter covers the theoretical foundations for the architectures developed in this thesis. We begin by formalizing single-agent Markovian decision processes and policy optimization. We then extend these concepts to multi-agent environments under partial observability, detailing the Centralized Training with Decentralized Execution (CTDE) paradigm and Multi-Agent Proximal Policy Optimization (MAPPO). Finally, we examine temporal abstraction frameworks, specifically Potential-Based Reward Shaping (PBRS) and the Options framework, before reviewing modern integrations of Large Language Models (LLMs) as cognitive controllers.

2.1 Fundamentals of Reinforcement Learning

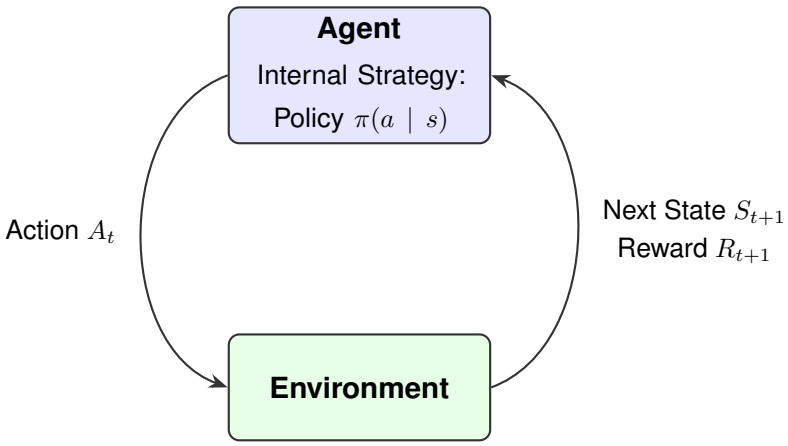
Reinforcement learning (RL) formalizes sequential decision-making under uncertainty, where an autonomous agent learns to optimize its behavior through interaction with a dynamic environment (Sutton et Barto 2018). Imagine an agent dropped into a complex grid-world, tasked with navigating a forest to collect wood. At each step, it observes its surroundings, chooses a direction, and receives a numerical reward if it successfully harvests the resource.

2.1.1 Markov Decision Processes (MDPs)

A fully observable single-agent control problem is mathematically formalized as a Markov Decision Process (MDP) (Szepesvári 2011). Conceptually, an MDP provides a structured framework for modeling decision-making in environments where outcomes are partly random and partly under the control of the agent.

This foundational interaction is depicted in Figure 1. At each discrete timestep, the agent observes the current state of the world, consults its internal strategy (the policy π) to select an action, and executes it. The environment then computes the consequences, responding with the subsequent state and a scalar reward signal.

Figure 1: The standard Reinforcement Learning interaction loop modeling an MDP.



To solve this grid-world mathematically, the system relies on the *Markov Property*. This property asserts a very simple rule: the future depends strictly on the present, not the past. If the agent knows its current state (where it is right now), memorizing the entire path it took to get there provides no additional advantage. Formally, the transition to the next state is independent of

historical trajectories:

$$\mathbb{P}[S_{t+1} = s' \mid S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, \dots] = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a] \tag{1}$$

Building on this property, the MDP is formally defined by the 5-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ (Szepesvári 2011). We can map each of these mathematical variables directly to our wood-collecting agent:

- $\mathcal{S}$ is the set of all valid states. In our example, a state s includes the agent's (x, y) coordinates on the grid and whether its inventory currently contains wood.
- $\mathcal{A}$ is the set of available actions. For our agent, this might be limited to {Move North, Move South, Interact/Chop}.
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability. If the agent chooses the 'Move North' action, $\mathcal{P}$ defines the chance it actually lands on the tile above (e.g., 100%, or perhaps 90% if the environment includes slippery terrain).
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function. This is the environment's feedback, such as emitting $+1.0$ for successfully chopping the wood, or -0.01 for wasting a step.
- $\gamma \in [0, 1)$ is the discount factor. This is a design parameter (e.g., 0.99) that makes immediate rewards slightly more valuable than future rewards, forcing the agent to collect the wood efficiently rather than wandering the grid forever.

To navigate this world, the agent needs a strategy, known as a *policy*. Rather than a rigid set of if/then rules, modern RL utilizes a stochastic (probabilistic) policy, denoted mathematically as $\pi(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s]$ (Sutton et Barto 2018). Intuitively, this means that given a specific state s, the policy outputs a probability distribution over all possible actions. For instance, if the agent is standing directly next to a tree, its policy π might assign a 95% probability to the 'Chop' action, and a 5% probability to 'Move South' to encourage a slight amount of random exploration.

The ultimate goal of the agent is to optimize this policy to maximize its *expected discounted return*, denoted as G_t (Sutton et Barto 2018). $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ represents the agent's "total lifetime score." It is the sum of every reward the agent expects to collect from the current moment t until the end of the episode, with future rewards multiplied by the discount factor γ so they carry slightly less weight than immediate victories.

To mathematically bind all the components of the MDP tuple together, we define the *Value function* $V^\pi(s)$. This function represents the expected return when starting in state s and following policy π thereafter. Crucially, it can be decomposed recursively into the *Bellman Equation* (Bellman 1957), which explicitly unites the policy (π) with the transition probabilities ($\mathcal{P}$), rewards ($\mathcal{R}$), and discount factor (γ) across the state-action space ($\mathcal{S}, \mathcal{A}$):

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s, a) \Big[\mathcal{R}(s, a) + \gamma V^\pi(s') \Big] \tag{2}$$

This equation is the engine of reinforcement learning. It demonstrates that the abstract variables of the MDP tuple do not exist in isolation, but interact continuously to evaluate and improve the agent's strategy. It states that the value of being in a state is equal to the immediate reward received plus the value of being in the next state, discounted by gamma.

2.1.2 The Actor-Critic Architecture: Two Neural Networks

To train our wood-collecting agent, we must optimize its policy. Instead of memorizing a rigid table of every possible grid coordinate, modern reinforcement learning parameterizes the policy using a neural network. This network, commonly referred to as the *Actor*, is denoted as $\pi_\theta(a \mid s)$. It outputs a probability distribution over available actions a given the current state s, utilizing its trainable weights $\theta \in \mathbb{R}^d$ (Sutton et Barto 2018).

To help the Actor learn, we introduce a second neural network called the *Critic*, parameterized by an entirely separate set of weights ϕ. The Critic learns the *Value function*, $V_\phi(s)$, which is its estimate of the total expected return from a given state. Conceptually, this is the agent's "gut feeling" about a location. If the agent stands next to a forest, the Critic outputs a high value; if it stands next to a hazard, it outputs a low value.

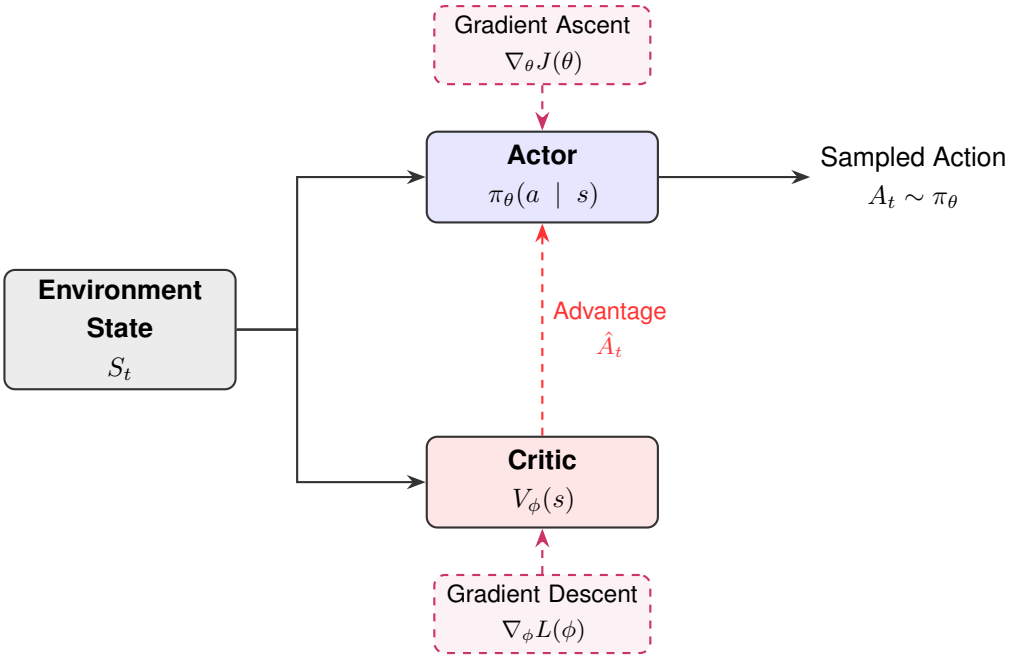
Because we have two distinct networks, they have two distinct mathematical objectives during optimization:

- **The Actor** aims to maximize its expected performance score, denoted as $J(\theta)$. Therefore, it updates its weights θ via *gradient ascent* (∇_θ). Shown later in Equation (5) and Figure 2.
- **The Critic** aims to minimize its prediction errors. Its objective is a loss function $L(\phi)$, typically defined as the Mean Squared Error between its predicted value and the actual target return $\hat{R}_t$. Therefore, it updates its weights ϕ via *gradient descent* (∇_ϕ):

$$L(\phi) = \hat{\mathbb{E}}_t \left[\frac{1}{2} \left(V_\phi(s_t) - \hat{R}_t \right)^2 \right] \tag{3}$$

As illustrated in Figure 2, this dual-network architecture creates a continuous feedback loop. The Critic evaluates the Actor's choice by generating an Advantage signal ($\hat{A}_t$), which dictates the direction of the Actor's policy gradient update, pushing it to favor successful actions in the future.

Figure 2: The Actor-Critic architecture.



2.1.3 The Critic's Evaluation: TD-Error and Advantage

Because the Critic's Value function is a prediction, it requires a mathematical mechanism to measure when it is wrong. This is achieved via the *Temporal Difference Error* (TD-Error), denoted as δ_t (Sutton et Barto 2018).

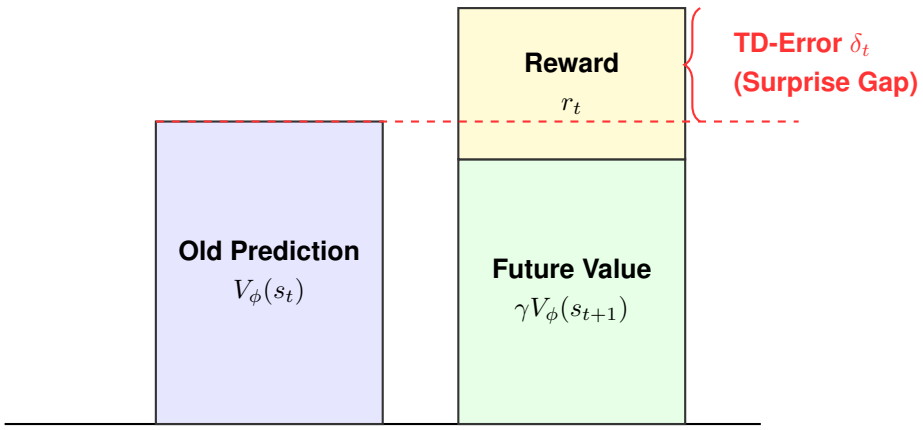
Let's consider our lumberjack agent. Suppose the Critic predicts that from its current state s_t, it expects to gather 50 future points ($V_\phi(s_t) = 50$). The agent then executes an action, immediately receives a small reward of 5 from the environment ($r_t = 5$), and lands in a new state s_{t+1}. The Critic evaluates this new state and predicts 45 future points ($V_\phi(s_{t+1}) = 45$).

The total updated estimate is the immediate reward plus the discounted future prediction: $r_t + \gamma V_\phi(s_{t+1})$. If we assume $\gamma = 1$ for simplicity, the new estimate is $5 + 45 = 50$. Because the new estimate perfectly matches the old prediction ($50 = 50$), the TD-Error is exactly 0. The Critic was perfectly accurate.

However, if the agent unexpectedly finds a hidden cache of wood and receives an immediate reward of 20, the new estimate becomes $20 + 45 = 65$. The TD-Error is the gap between reality and expectation: $\delta_t = 65 - 50 = +15$. This positive "surprise" signal forces the Critic to update its weights ϕ to make better predictions in the future. We can visualize this mismatch in Figure 3. Formally, the TD-Error is defined as:

$$\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \tag{4}$$

Figure 3: Visualizing the Temporal Difference (TD) Error as the mathematical gap between the Critic's prior prediction and the newly observed reality.



Building upon this, we must evaluate the specific action the Actor chose. We define the state-value $V^{\pi_\theta}(s)$ as the baseline expectation of simply *being* in a state. Conversely, we define the action-value $Q^{\pi_\theta}(s, a)$ as the expected return of taking a *specific action* in that state before resuming the regular policy.

The *Advantage* function, $A^{\pi_\theta}(s, a)$, isolates the unique benefit of that specific action by subtracting the baseline from the action-value: $A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s)$ (Sutton et Barto 2018). Modern algorithms estimate this Advantage using a discounted sum of the TD-Errors we just calculated.

2.1.4 The Actor's Update and Proximal Policy Optimization (PPO)

With the Advantage calculated by the Critic (Figure 2), the Actor knows whether its chosen action was better or worse than expected. Classically, by the Policy Gradient Theorem (Sutton et Barto 2018), the Actor updates its weights θ to maximize its performance via the following gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^{\pi_\theta}, a\sim\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a \mid s) A^{\pi_\theta}(s,a)\right] \tag{5}$$

Mathematically, this equation states that the gradient of the Actor's overall performance ($\nabla_\theta J(\theta)$) is equal to an expected value (an average) taken over recent experiences. For every state-action pair, we calculate the gradient of the log probability ($\nabla_\theta \log \pi_\theta$), the exact direction we must shift our weights to make that specific action more likely, and scale it by the Critic's Advantage signal (A^{π_θ}). In plain English, Equation (5) tells the neural network: *"On average across all your recent experiences, if an action was surprisingly good (positive Advantage), push your weights in the direction that makes that action more likely in the future. If it was bad (negative Advantage), push in the opposite direction."*

We can see why it could be fragile in practice. Suppose our lumberjack agent accidentally executes a random sequence of movements and luckily discovers a massive, hidden cache of wood. The Critic will calculate an enormous positive Advantage. Standard gradient ascent will blindly follow the math, reacting by violently dragging the Actor's weights in that specific direction to favor that exact sequence again.

This aggressive update frequently causes the network to destabilize. By over-updating its internal structure for one lucky event, the agent effectively overwrites its learned weights, forgetting fundamental survival rules it had previously learned (such as avoiding hazards or map boundaries).

Proximal Policy Optimization (PPO) resolves this fragility. It abandons the classical gradient calculation and instead enforces a strict mathematical guardrail, establishing a clipped surrogate objective specifically for the Actor, denoted as $L^{\text{CLIP}}(\theta)$ (Schulman et al. 2017):

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min\left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \tag{6}$$

To fully understand this equation, we can break down its notation:

- $\hat{\mathbb{E}}_t$ means we are averaging over a finite batch of real data collected at time steps t, rather than computing a perfect theoretical expectation. $\hat{A}_t$ is the estimated Advantage provided by the Critic.
- $r_t(\theta)$ is the probability ratio, defined as $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$, which measures how much more (or less) likely the agent is to take the action now compared to before the update.
- ϵ acts as a hard boundary or a speed limit for the updates (typically 0.2, as established by Schulman et al. (2017)).

If an action yields a massive positive Advantage, the network will naturally try to increase the ratio $r_t(\theta)$ exponentially. However, the clip function truncates this ratio at $1+\epsilon$ (i.e., 1.2). The $\min$ operator then forces the optimizer to use this truncated version, completely ignoring any

further gradient once the probability of that action has increased by 20%. By mathematically ignoring overly enthusiastic updates, PPO ensures the Actor learns smoothly and stably without destructive variance spikes.

2.1.5 Kullback-Leibler (KL) Divergence and Early Stopping

While the clipping mechanism described in Equation 6 acts as a strict per-action speed limit, ensuring the probability of taking one specific action doesn't spike too drastically, PPO implementations frequently employ an additional, broader safety net. This is known as *early stopping* based on Kullback-Leibler (KL) divergence (Schulman et al. 2017).

To understand why this is necessary, consider our lumberjack once more. The clipping function works perfectly for isolating and limiting a single enthusiastic update (like chopping a tree). However, if the agent updates thousands of small, unclipped actions simultaneously across a large batch of data, the *cumulative* effect might still fundamentally alter the agent's overall behavior.

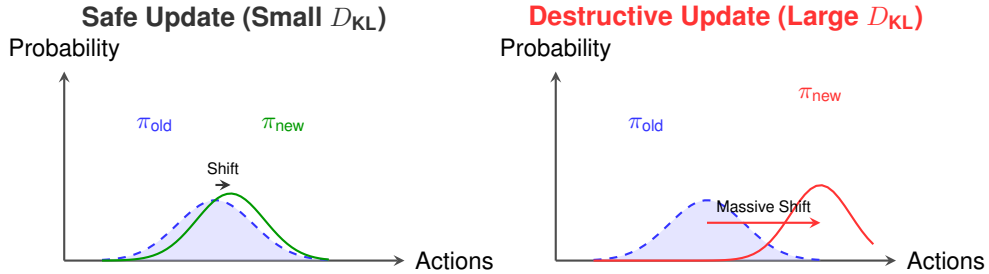
KL divergence is a concept from information theory that measures exactly this: how much one probability distribution diverges from a reference distribution. In our case, it measures the mathematical "distance" between the agent's new updated policy $\pi_{\theta_{\text{new}}}$ and its old policy $\pi_{\theta_{\text{old}}}$ before the current optimization step began. For discrete action spaces, it is defined as:

$$D_{\text{KL}}(\pi_{\theta_{\text{old}}} \parallel \pi_{\theta_{\text{new}}}) = \sum_{a \in \mathcal{A}} \pi_{\theta_{\text{old}}}(a \mid s) \log \left(\frac{\pi_{\theta_{\text{old}}}(a \mid s)}{\pi_{\theta_{\text{new}}}(a \mid s)} \right) \tag{7}$$

Importantly, KL divergence is asymmetric (i.e., $D_{\text{KL}}(P \parallel Q) \neq D_{\text{KL}}(Q \parallel P)$), meaning it specifically measures the information lost when the new policy is used to approximate the old policy's distribution. It is always non-negative and equals exactly zero only when the two policies are identical.

Let us consider the lumberjack standing next to a tree. Under the old policy, it might have a 90% probability of choosing to 'Chop' and a 10% probability of choosing to 'Move'. If a single update drastically shifts these probabilities to 10% 'Chop' and 90% 'Move', the KL divergence between the two policies will spike. As illustrated in Figure 4, a massive spike indicates that the optimizer has aggressively destroyed the previously learned behavior, potentially causing the agent to forget fundamental skills.

Figure 4: Visualizing KL Divergence. A small divergence (left) preserves the core behavior while optimizing safely. A large divergence (right) indicates the new policy has aggressively abandoned the old behavior, potentially leading to catastrophic forgetting.



To prevent this catastrophic forgetting, PPO calculates an approximation of the KL divergence during the training loop. If this calculated distance crosses a strict, predefined threshold (the Target KL, typically around 0.01 to 0.015), the optimizer acts as an emergency brake. It immediately aborts any remaining training epochs for that batch of data (Schulman et al. 2017). This ensures that even if a wave of subtle updates slips past the cruise-control of the per-action clipping mechanism, the agent's "brain" is physically blocked from mutating too violently in a single step.

2.2 Cooperative Multi-Agent Reinforcement Learning

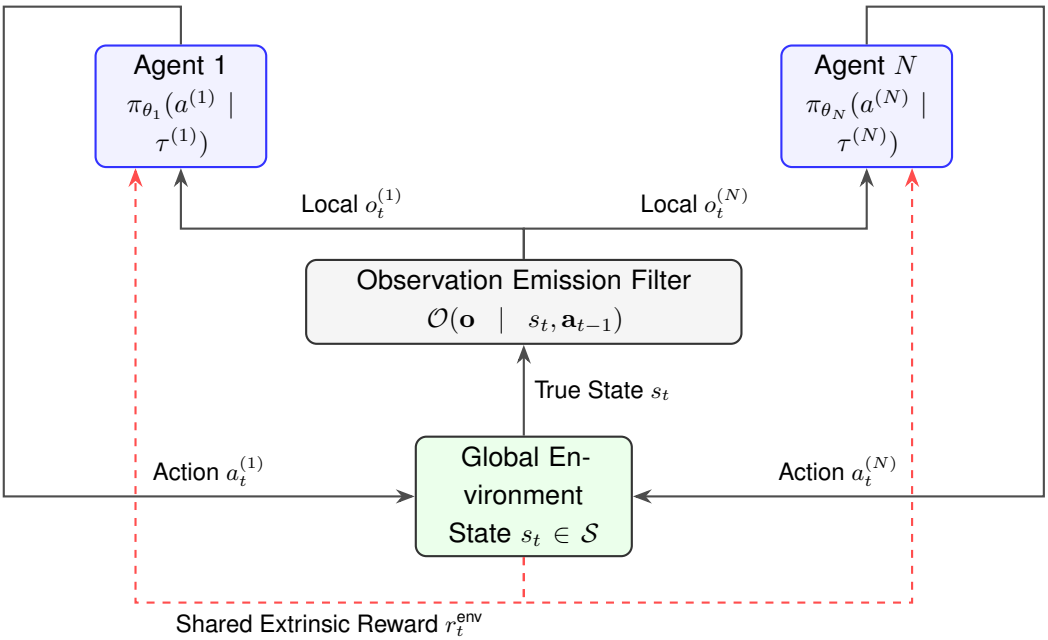
Imagine our wood-collecting agent is no longer alone. We now introduce a second agent into the grid, tasked with mining stone. Suddenly, the fundamental assumption of our standard MDP (that the environment only changes when *our* agent acts) is invalidated. Because both agents update their policies and take actions concurrently, the environment becomes a moving target. If the lumberjack moves toward a tree, but the miner accidentally blocks the path a millisecond prior, the environment's transition dynamics become non-stationary from the local perspective of any individual agent (Albrecht, Christianos et Schäfer 2024).

2.2.1 Decentralized Partially Observable MDPs (Dec-POMDPs)

In complex multi-agent domains, an individual agent rarely possesses a god's-eye view of the entire grid. They suffer from local sensory limitations, perceiving only a cropped window of their immediate surroundings.

As illustrated in Figure 5, this scenario is formally modeled as a Decentralized Partially Observable Markov Decision Process (Dec-POMDP) (Oliehoek et Amato 2016). The diagram demonstrates how the true global environment state is hidden behind an "observation emission filter," essentially creating strict partial observability that forces each agent to act strictly on fragmented, local information.

Figure 5: The Dec-POMDP interaction framework.



To formalize this mathematically, we extend our previous MDP into the Dec-POMDP tuple $\langle \mathcal{N}, \mathcal{S}, \boldsymbol{\mathcal{A}}, \mathcal{P}, \mathcal{R}, \boldsymbol{\Omega}, \mathcal{O}, \gamma \rangle$. We can map these variables directly to our cooperative scenario:

- $\mathcal{N} = \{1, \dots, N\}$ is the roster of active agents (e.g., Agent 1 Lumberjack, Agent 2 Miner).
- $\boldsymbol{\mathcal{A}} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_N$ is the joint action space. At timestep t, agents simultaneously execute a joint action $\mathbf{a}_t = (a_t^{(1)}, \dots, a_t^{(N)}) \in \boldsymbol{\mathcal{A}}$.
- $\boldsymbol{\Omega} = \Omega_1 \times \cdots \times \Omega_N$ is the joint observation space, representing the limited windows of information each agent receives.
- $\mathcal{O} : \mathcal{S} \times \boldsymbol{\mathcal{A}} \times \boldsymbol{\Omega} \to [0, 1]$ is the observation probability distribution $\mathcal{O}(\mathbf{o} \mid s', \mathbf{a})$. This acts as the mathematical "filter" shown in Figure 5, determining what slice of the true state each agent is permitted to see.
- $\mathcal{R} : \mathcal{S} \times \boldsymbol{\mathcal{A}} \to \mathbb{R}$ is the globally shared reward signal. If the agents successfully cooperate to build a tool, this single scalar reward is emitted to all agents equally.

This structure introduces a critical mathematical consequence. Because an individual agent i cannot perceive the full state s_t, a single observation $o_t^{(i)} \in \Omega_i$ instantly breaks the Markov property we established in Section 2.1.1. The agent cannot rely solely on the present moment, because its present observation is incomplete.

To restore temporal context, agent i must rely on its memory. It maintains an action-observation history $\tau_t^{(i)} = (o_1^{(i)}, a_1^{(i)}, \dots, o_t^{(i)})$, and parameterizes its decentralized policy as $\pi_i(a_i \mid \tau_i)$. While standard feed-forward or Convolutional Neural Networks (CNNs) (Lecun et al. 1998) are highly effective at extracting spatial features from a single isolated observation, they inherently lack the capacity to remember past states. This is why modern architectures implement this memory using Recurrent Neural Networks (RNNs). Gated variants like Long Short-Term Memory (LSTM) (Hochreiter et Schmidhuber 1997) or Gated Recurrent Units (GRUs) (Chung et al. 2014) are typically used to prevent the vanishing gradient problem. By maintaining stable learning signals over time, these networks allow the agent to build a continuous understanding of the environment from fragmented spatial snapshots (Hausknecht et Stone 2015).

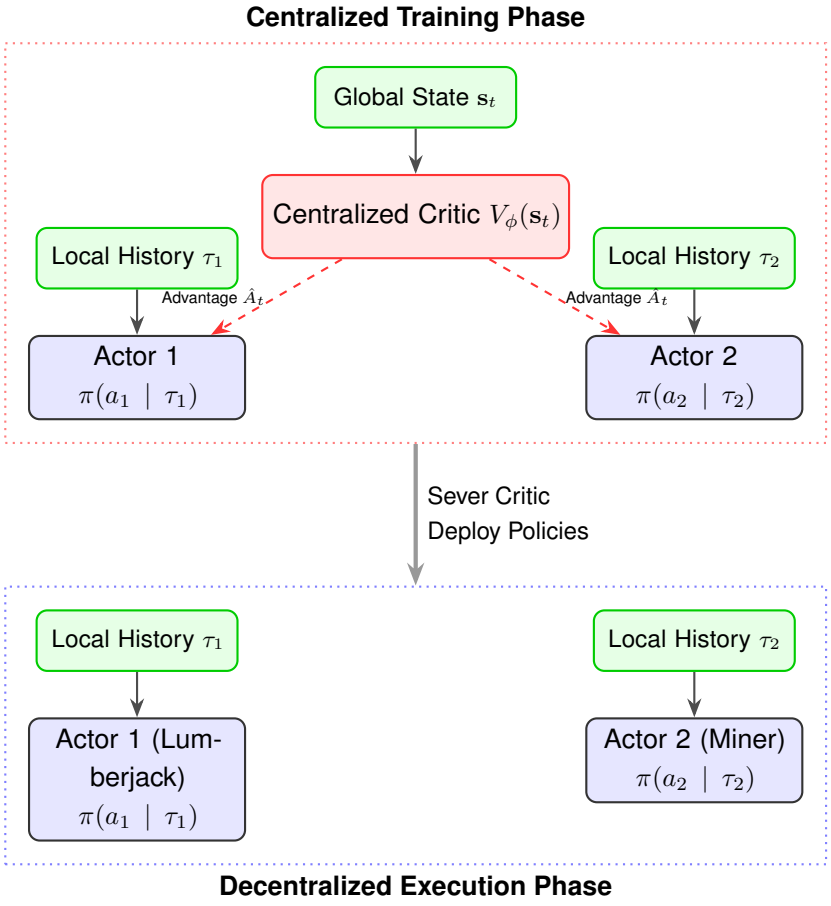
2.2.2 Centralized Training with Decentralized Execution (CTDE)

To resolve this partial observability without violating the strict rules of the game, modern MARL algorithms adopt the Centralized Training with Decentralized Execution (CTDE) paradigm (Albrecht, Christianos et Schäfer 2024).

CTDE applies a dual-perspective philosophy to our grid-world. As shown in Figure 6, during the offline training phase, the algorithm acts as an omniscient supervisor. It utilizes a Centralized Critic that has complete access to the global environment state $\mathbf{s}_t$. Because it knows the exact coordinates of both the lumberjack and the miner, it can accurately judge whether a joint action was truly beneficial.

However, during online deployment, this centralized structure is severed. The Critic is discarded, and the decentralized Actors are deployed to the grid. They must execute their policies independently, relying solely on their fragmented local action-observation histories (τ_i) without knowing the exact status of their partner.

Figure 6: The Centralized Training with Decentralized Execution (CTDE) architecture.



Adapted from (Yu et al. 2022)

2.2.3 Multi-Agent Proximal Policy Optimization (MAPPO)

Before the widespread adoption of Actor-Critic methods in multi-agent settings, the field heavily relied on value-decomposition techniques to solve the credit assignment problem. Algorithms like Value-Decomposition Networks (VDN) (Sunehag et al. 2017) and QMIX (Rashid et al. 2018) became popular by factorizing a global reward into individual agent contributions. QMIX, for example, enforces a strict monotonic relationship between the global value and the local values of the agents. However, these structural constraints limit the types of cooperative tasks these algorithms can solve effectively (Castellini 2022).

More recently, research has demonstrated that on-policy Actor-Critic methods can avoid these constraints altogether while matching or exceeding the performance of off-policy factorization methods. Multi-Agent PPO (MAPPO) brings our grid-world to life by extending the single-agent PPO architecture directly into the CTDE paradigm (Yu et al. 2022). MAPPO is straightforward: the decentralized Actors remain mathematically identical to standard PPO, while only the Centralized Critic is modified to process the global environment state.

The Centralized Critic Network (Value), parameterized by ϕ, learns the value function $V_\phi(\mathbf{s}_t)$. Crucially, $\mathbf{s}_t$ is the unclipped global joint state. In our RPG scenario, while the lumberjack only sees the trees immediately in front of them, the Critic sees the entire map, knowing exactly where both the lumberjack and the miner are located, and what resources they hold. By estimating baseline returns using this true reality of the board, the Critic stabilizes the variance in its

Advantage calculations. It is optimized by minimizing the Mean Squared Error against the actual calculated returns $\hat{R}_t$ (Yu et al. 2022):

$$L^{\mathsf{VF}}(\phi) = \frac{1}{B \cdot N} \sum_{i=1}^{B} \sum_{k=1}^{N} \left(V_\phi(\mathbf{s}_{t,k}) - \hat{R}_{t,k} \right)^2 \tag{8}$$

where N represents the number of agents, and B represents the batch size (the number of environment transitions sampled during a training update). This equation simply averages the prediction error across all agents and all sampled experiences.

Simultaneously, the decentralized Actors (the lumberjack and the miner) learn their respective policies $\pi_{\theta_i}(a_i \mid \tau_i)$. Because they operate under strict partial observability, their inputs are restricted strictly to their localized action-observation histories (τ_i). Each Actor updates its weights using the exact same clipped surrogate objective from Equation 6.

As shown in Figure 6, the gradient flow is strictly separated. The global state error updates only the Critic's weights (ϕ), while the Critic passes its computed Advantage signal to the Actors. The lumberjack and miner then use this Advantage to update their own weights (θ_i). This clear separation enables MAPPO to achieve sample efficiency in complex cooperative benchmarks (Yu et al. 2022).

2.3 Curriculum Learning

Some environments suffer from extremely sparse rewards, meaning agents only receive feedback after completing a long sequence of actions. When this happens, an alternative to altering the agent's internal architecture is to alter the environment itself. Bengio et al. (2009) formalized this concept as *Curriculum Learning*, drawing inspiration from human education.

Rather than training a model on the most difficult data (or the most complex RL environment) from the start, the agent is first exposed to simplified, easily solvable versions of the task. As the agent masters these basic concepts, the environment's complexity is progressively increased. In reinforcement learning, this typically means a human designer must manually craft a sequence of environments. For example, starting the agent in a tiny grid with the target one step away, and gradually expanding the grid size and adding obstacles.

Although effective, this traditional approach suffers from scalability issues due to its reliance on manual engineering. To overcome this, the field shifted toward *Automated Curriculum Learning* (ACL), where algorithms dynamically adjust the environment parameters (Portelas et al. 2020).

However, generating entirely new physical environments on the fly introduces significant computational overhead and non-stationarity. If the physical rules or map layouts constantly change, the agents struggle to stabilize their policies. Therefore, rather than altering the environment from the outside, a more stable solution is to alter the agent's cognitive abstractions from the inside. By shaping the internal rewards or providing discrete skill milestones, techniques explored in the following section, we can guide agents through complex dependencies without ever needing to rebuild the world around them.

2.4 Temporal Abstraction and Reward Shaping

While MAPPO exhibits strong sample efficiency in dense tasks, it struggles significantly in long-horizon domains gated by sparse rewards. If our lumberjack and miner only receive a single $+10$ reward upon completing a high-level goal, such as successfully crafting an advanced tool, the probability of them randomly discovering the exact prerequisite sequence of chopping wood, mining stone, and navigating to a workbench approaches zero. To mitigate this exploration bottleneck, structural abstraction techniques are required.

2.4.1 Potential-Based Reward Shaping (PBRS)

Reward shaping attempts to solve this exploration problem by supplementing the sparse environmental reward r^{env} with an auxiliary dense signal $F(s, s')$, gently guiding the agent toward the desired objective. However, unconstrained reward additions frequently alter the underlying MDP, causing agents to discover local optima or engage in reward hacking (e.g., farming points by repeating an action without actually progressing).

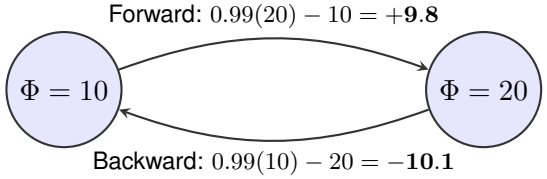
To prevent this policy distortion, Ng et al. (1999) proved that a shaping signal $F(s, s')$ preserves the optimal policy set π^* if and only if it is expressed as the difference of a state potential function $\Phi(s)$ (Ng, Harada et Russell 1999):

$$F(s, s') = \gamma\Phi(s') - \Phi(s) \tag{9}$$

where the potential function $\Phi : \mathcal{S} \to \mathbb{R}$ evaluates any state and outputs a scalar score. In our grid-world, think of $\Phi(s)$ as a literal hot-and-cold compass pointing the lumberjack toward the nearest tree. Adding $F(s, s')$ to the environment reward creates an augmented, safe signal $r_t^{\text{total}} = r_t^{\text{env}} + F(s_t, s_{t+1})$.

As demonstrated in Figure 7, the discount factor γ is mathematically vital. It acts as a friction mechanism, taxing the agent's movement. When the agent steps forward to a higher potential state, the reward is slightly discounted. However, when it steps backward to a lower potential state, the penalty outweighs the previous gain. Because any cyclic loop yields a net negative return (e.g., $\sum_{t=0}^{\infty} \gamma^t F(s_t, s_{t+1}) = -\Phi(s_0)$), this formulation guarantees policy invariance while providing dense guidance (Ng, Harada et Russell 1999).

Figure 7: The anti-farming mechanism of PBRS. The discount factor $\gamma = 0.99$ ensures that returning to a previous state penalizes the agent more heavily than the forward step rewarded it. This yields a net loss for cyclic behavior (e.g., $+9.8 - 10.1 = -0.3$), preventing infinite reward loops.



2.4.2 The Options Framework for Hierarchical RL

Rather than modifying the reward landscape, Hierarchical Reinforcement Learning (HRL) achieves temporal abstraction by organizing low-level actions into high-level macro-structures. The *Options Framework* formalizes these macro-actions over MDPs (Sutton, Precup et Singh 1999). Rather than forcing a single policy to decide whether to step north or south at every millisecond, HRL introduces a higher-level cognitive hierarchy. A Meta-Controller can select a discrete Option $o \in \mathcal{O}$, such as `COLLECT_WOOD`, which then dictates those low-level kinetic movements.

This Option is defined by the triple $\langle \mathcal{I}_o, \pi_o, \beta_o \rangle$ (Sutton, Precup et Singh 1999):

- $\mathcal{I}_o \subseteq \mathcal{S}$ is the initiation set. The `COLLECT_WOOD` Option can only be initiated if the lumberjack does not already have wood in its inventory.
- $\pi(a \mid s, o)$ is the intra-option policy governing primitive execution. While classically denoted as a separate policy $\pi_o(a \mid s)$, modern deep RL implementations parameterize this as a single neural network $\pi_\theta(a \mid s, o)$ that takes the selected option o as an augmented input. This is the low-level MAPPO policy that actually steers the agent to the tree and executes the 'Chop' command.
- $\beta_o : \mathcal{S} \to [0, 1]$ is the stochastic termination condition, giving the probability $\beta_o(s)$ that option o terminates in state s. For the lumberjack, this triggers at 100% probability the moment the wood enters the inventory.

When an option o is selected, the intra-option policy π_o controls primitive actions until the termination condition β_o triggers. This transforms the underlying decision process into a Semi-Markov Decision Process (SMDP) operating over variable temporal steps (Sutton, Precup et Singh 1999).

Because an Option o executes over a variable number of steps k rather than a single discrete tick, the standard MDP mathematics must be adjusted. The Meta-Controller evaluates its strategic choices using an SMDP action-value function. The reward is accumulated over the duration of the option, and the value of the resulting next state S_{t+k} is discounted by γ^k:

$$Q(s, o) = \mathbb{E}\left[\sum_{j=1}^{k} \gamma^{j-1} R_{t+j} + \gamma^k \max_{o'} Q(S_{t+k}, o')\right] \tag{10}$$

The total value of choosing an Option ($Q(s, o)$) is the sum of all the small environmental rewards collected *during* its execution (the $\sum$ term), plus the expected value of the new state we land in once the option finishes (S_{t+k}). Because this new state is reached k steps in the future, its value is exponentially discounted by γ^k. This formulation is important for Hierarchical RL: it empowers the high-level controller to plan over long semantic horizons while remaining robust to the unpredictable amount of physical time (k) required by the low-level agents.

Crucially, this temporal abstraction replaces tick-by-tick micromanagement. Because the high-level controller only evaluates the environment when an Option terminates, it operates at a significantly lower frequency than the environment's physical clock (Sutton, Precup et Singh 1999). By abstracting the kinetic physics of the grid into discrete semantic tasks, the Options framework sets the stage for a high-level cognitive controller. As explored in the next chapter, this

architecture allows computationally expensive Large Language Models to act as these strategic orchestrators without needing to generate a decision every physical step, entirely bypassing the random exploration bottleneck.

3 Background: Large Language Models and Fine-Tuning

In the previous chapter, we established two methods to guide MARL agents through sparse domains: manipulating their continuous reward signals via Potential-Based Reward Shaping, or organizing their movements into discrete macro-actions via the Options framework. However, MARL agents cannot inherently deduce the optimal shaping parameters or the logical sequence of these macro-actions on their own. This chapter introduces the theoretical foundations of Large Language Models (LLMs) and their optimization mechanisms. It establishes how these autoregressive models can act as a high-level "Commander," parsing the semantic state of the grid-world to either continuously tune the reward shaping mixing board or issue discrete strategic sequences to the agents below.

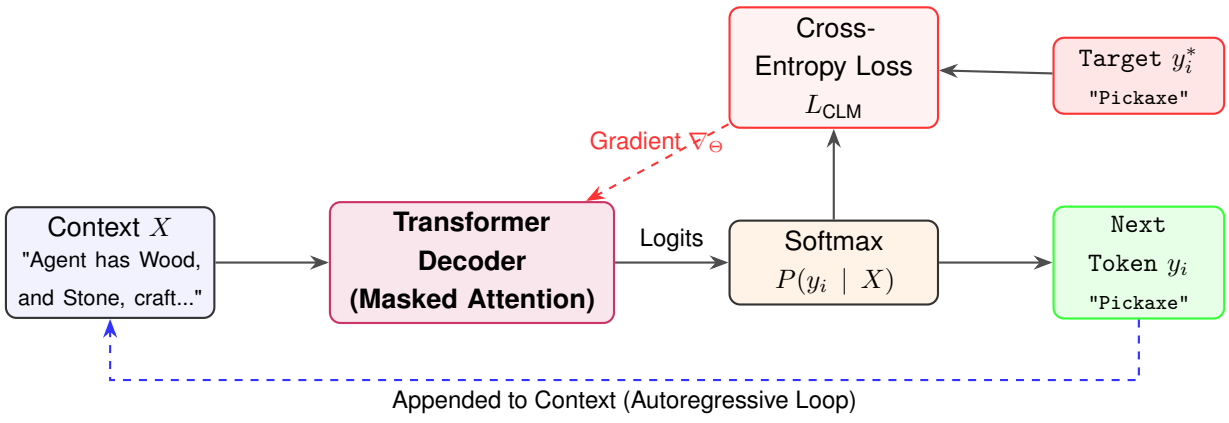
3.1 Foundations of Causal Language Modeling (CLM)

While *Causal Language Models* (CLMs) were originally introduced for predicting text sequences in natural language understanding tasks (Radford et al. 2018), their autoregressive nature has been framed as an ideal architecture for processing a current state and executing sequential decisions (Chen et al. 2021).

Unlike standard RL policies that output continuous spatial coordinates, CLMs operate over massive discrete vocabularies. They generate text autoregressively, meaning they predict the next logical token (word or sub-word) based strictly on the sequence of past tokens. This step-by-step generation perfectly mirrors the step-by-step decision-making process of an RL agent.

Modern CLMs, including the Qwen architecture utilized in this thesis (Yang et al. 2025), rely entirely on the Transformer architecture (Vaswani et al. 2017). To understand this conceptually, imagine our Commander receives a text context: *"Agent has Wood and Stone, craft a..."* As illustrated in Figure 8, to accurately predict the next target token ("Pickaxe"), the network must understand the relationship between "Wood", "Stone", and "craft" despite the distance between these words in the sentence. Transformers achieve this by dispensing with temporal recurrence in favor of parallelizable *self-attention* mechanisms.

Figure 8: The Causal Language Modeling (CLM) autoregressive generation and training loop.



3.1.1 Scaled Dot-Product Attention and Causal Masking

The core computational engine that allows the model to connect these distant concepts is Scaled Dot-Product Attention. For a given input sequence, the model embeds the tokens into three distinct weight matrices: Queries (Q), Keys (K), and Values (V) (Vaswani et al. 2017).

- **Query (Q):** What the current token is looking for (e.g., the word "craft" sends a query searching for materials).
- **Key (K):** What previous tokens contain (e.g., "Wood" and "Stone" hold keys identifying them as available materials).
- **Value (V):** The actual semantic content extracted if the Query and Key match.

The attention mechanism computes a weighted sum of these values, where the weight is determined by how well the query and the key match via a dot-product:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V = WV \tag{11}$$

There are two critical mathematical safeguards in this equation. First is the scaling factor, $\sqrt{d_k}$ (the square root of the key vector dimension). If Q and K are large, their dot product grows massive. Feeding massive numbers into a softmax function causes it to saturate, snapping its outputs to absolute 1s and 0s. This flattens the mathematical slope (the gradient) to zero, halting the network's ability to learn, a phenomenon known as the vanishing gradient problem. Dividing by $\sqrt{d_k}$ cools these values down, preserving the gradient.

Second, because we are using a *Causal* Language Model, the network must not be allowed to cheat by looking into the future. During training, the entire text sequence is fed into the Transformer simultaneously for parallel processing efficiency. In a standard full-attention mechanism, the QK^T matrix computes the dot-product relationship between every token and every other token all at once. If our sequence is "Agent has Wood and Stone", the first token "Agent" would mathematically be able to query the future token "Wood" to help predict the second token "has". This would give the model an impossible advantage during training that it would not have during real-time generation.

To prevent this information leak, a strict lower-triangular mask is applied to the QK^T matrix before the softmax function. This mask overrides the raw attention scores of all future tokens with negative infinity ($-\infty$). When the softmax function is applied, the exponential function $e^{-\infty}$ evaluates to exactly zero, completely severing the connection to future words.

Visually, for a sequence of 4 tokens, we can isolate the intermediate weight matrix W from Equation 11 (specifically, the softmax $\left(\frac{QK^T}{\sqrt{d_k}}\right)$ component) to see how the causal mask operates before the final multiplication by the Values (V):

$$W = \text{softmax}\left(\begin{bmatrix} s_{1,1} & -\infty & -\infty & -\infty \\ s_{2,1} & s_{2,2} & -\infty & -\infty \\ s_{3,1} & s_{3,2} & s_{3,3} & -\infty \\ s_{4,1} & s_{4,2} & s_{4,3} & s_{4,4} \end{bmatrix}\right) = \begin{bmatrix} w_{1,1} & 0 & 0 & 0 \\ w_{2,1} & w_{2,2} & 0 & 0 \\ w_{3,1} & w_{3,2} & w_{3,3} & 0 \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} \end{bmatrix} \tag{12}$$

where $s_{i,j}$ represents the raw, scaled dot-product score between query i and key j (an element of the $\frac{QK^T}{\sqrt{d_k}}$ matrix), and $w_{i,j}$ represents the final valid probability weight. As shown in Equation 12, when the model is processing token 3 (row 3), it can only attend to past and current tokens 1, 2, and 3. The connection to the future token 4 is mathematically erased (0). This resulting weight matrix W is then multiplied by the Value matrix V to produce the final output of the attention mechanism.

3.1.2 The Autoregressive Loop and Optimization

As we have seen, the attention mechanism relies on a *softmax* function to convert raw query-key scores into valid probability weights. However, this is not the only place softmax is used. At the very end of the network, a separate, final softmax layer is required to convert the model's output logits into the probability distribution $P(y_i)$ of the next token over the entire vocabulary (as seen in Figure 8).

The generation process behaves like a strict deterministic time-loop: the model predicts a single word, and that new word instantly becomes part of the unchangeable past context used to predict the very next word (as shown by the blue feedback loop in Figure 8). Although we do not train a model from scratch in our implementation, it is relevant to understand the training objective. Given a context sequence X and a target sequence of tokens $\mathbf{y} = (y_1, y_2, \ldots, y_N)$, the network is optimized by minimizing the negative log-likelihood (Categorical Cross-Entropy) over the sequence (Radford et al. 2018):

$$L_{\text{CLM}} = -\frac{1}{N}\sum_{i=1}^{N} \log P(y_i \mid y_{<i}, X; \Theta) \tag{13}$$

where Θ represents the parameters of the neural network. As visualized by the backward gradient flow in Figure 8, it penalizes the model when the probability assigned to the correct target token y_i^* is too low, and gradients (∇_Θ) are backpropagated to update the model weights.

This pre-training over trillions of tokens gives LLMs implicit knowledge of logical dependencies, domain facts, and common-sense heuristics.

But why is this specific architecture so critical for our multi-agent framework? During a standard MARL episode, the local Qwen model acts as a frozen, inference-only "Commander". It is *not* actively learning or updating its weights via Equation 13. Instead, we rely entirely on the Attention Mechanism (Equation 11) to perform *in-context* reasoning. When we feed the model the full game state as a prompt (e.g., *"Agent 1 is at [2,3] and holds Wood and Stone. It must craft a Pickaxe,"*), the attention mechanism is exactly what allows the network to mathematically link the available inventory (*"Wood"*, *"Stone"*) with the goal (*"Pickaxe"*) to deduce the correct next macro-action. The autoregressive generation loop then allows it to construct and output the command `GO_TO_WORKBENCH` step-by-step.

However, as we will explore in the following section, relying purely on a base model's general knowledge is often insufficient for formatting precise JSON commands. We must first adapt this foundation using parameter-efficient fine-tuning (QLoRA) prior to deployment.

3.2 Parameter-Efficient Fine-Tuning (QLoRA)

While a base LLM understands general logic, it does not inherently know how to act as a programmatic controller. In our MARL environment, the Python simulation cannot parse a conversational paragraph; it requires the Commander's output to be a rigidly formatted JSON object. This structured format allows the environment to reliably sample and extract specific data points, such as priority weights for the Mixing Board, or categorical string options like `CRAFT_PICKAXE` for the Meta-Controller.

Teaching a base model these specific formatting rules via full parameter fine-tuning requires substantial compute clusters, rendering it infeasible for local execution alongside a MARL training loop. To mitigate this, this thesis utilizes Quantized Low-Rank Adaptation (QLoRA) (Dettmers et al. 2023). Conceptually, imagine the base LLM is a massive, immutable public library of knowledge. Rather than rewriting every book in the library, QLoRA simply slips a highly specialized, lightweight guidebook into the building that tells the model exactly how to format its thoughts into JSON for our grid-world.

3.2.1 Quantization and Low-Rank Matrices

Mathematically, this memory saving is achieved in two steps. First, QLoRA freezes the model's pre-trained weight matrix, $W_0 \in \mathbb{R}^{d \times k}$, using 4-bit NormalFloat (NF4) quantization (Dettmers et al. 2023; Hu et al. 2021). To understand quantization, imagine converting a high-resolution, 32-bit digital photograph into an 8-bit retro pixel image. While you lose microscopic precision, the overall shape and meaning of the image remain perfectly clear, and the file size shrinks drastically. NF4 quantization applies this exact compression to the neural weights, drastically reducing the VRAM required to hold the model's foundational knowledge in memory.

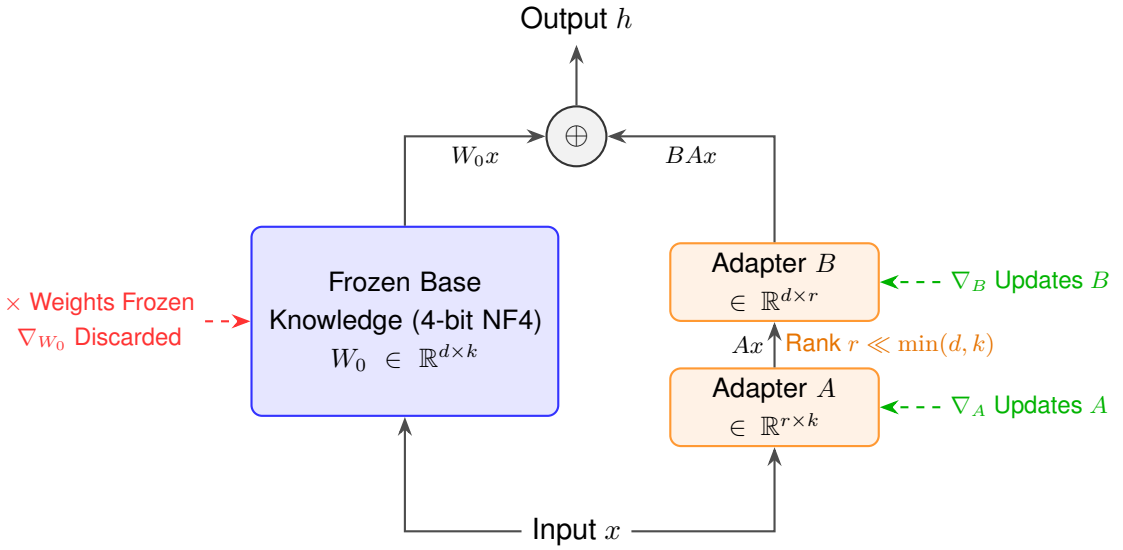
Second, because W_0 is now frozen and cannot be updated, we must inject trainable parameters. These form the specialized guidebook we mentioned. As illustrated in Figure 9, LoRA implements this "guidebook" by injecting two small, low-rank adapter matrices alongside the frozen base: matrix $A \in \mathbb{R}^{r \times k}$ and matrix $B \in \mathbb{R}^{d \times r}$ (Hu et al. 2021; Dettmers et al. 2023). During the fine-tuning process, the library (W_0) is left entirely alone, and *only* the weights within matrices A and B are updated via gradient descent.

To visualize why this is so efficient, consider a theoretical weight matrix W_0 with dimensions $10,000 \times 10,000$. Updating this full matrix would require learning 100,000,000 parameters. Instead, LoRA introduces a "bottleneck" defined by the rank r. If we choose a tiny rank, such as $r = 8$, matrix A becomes $8 \times 10,000$ (80,000 parameters) and matrix B becomes $10,000 \times 8$ (80,000 parameters). By multiplying them together ($B \times A$), we reconstruct a matrix ΔW that is still $10,000 \times 10,000$, but we only had to train 160,000 actual parameters (a reduction of over 99.8%). Equation 14 illustrates this bottleneck visually for a small 4×4 matrix with rank $r = 1$,

where training only 8 parameters is required to reconstruct all 16 weights:

$$\underbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}}_{B \in \mathbb{R}^{4\times 1}} \times \underbrace{\begin{bmatrix} a_1 & a_2 & a_3 & a_4 \end{bmatrix}}_{A \in \mathbb{R}^{1\times 4}} = \underbrace{\begin{bmatrix} b_1a_1 & b_1a_2 & b_1a_3 & b_1a_4 \\ b_2a_1 & b_2a_2 & b_2a_3 & b_2a_4 \\ b_3a_1 & b_3a_2 & b_3a_3 & b_3a_4 \\ b_4a_1 & b_4a_2 & b_4a_3 & b_4a_4 \end{bmatrix}}_{\Delta W \in \mathbb{R}^{4\times 4}} \tag{14}$$

Figure 9: The Low-Rank Adaptation (LoRA) architecture.



Adapted from (Hu et al. 2021)

3.2.2 The Forward Pass and Backpropagation

To ensure that these untrained adapters do not inject random noise into the network at the beginning of training, matrix A is initialized with random Gaussian values, while matrix B is initialized as a strict zero matrix (Hu et al. 2021). As a result, their initial product $BA = 0$.

During the forward pass (when the model is generating text), the input x flows through both the frozen base model and the new adapters. The modified forward pass is defined as:

$$h = W_0x + \Delta Wx = W_0x + \frac{\alpha}{r}(BA)x \tag{15}$$

where α is a constant scaling factor (Hu et al. 2021; Dettmers et al. 2023). Conceptually, α acts as a volume knob. It determines how "loudly" the new specialized JSON-formatting knowledge (BA) should be blended into the general world knowledge (W_0).

Once this forward pass is complete and the model generates its sequence, how does it actually learn to fix its mistakes? This is where the Causal Language Modeling loss (L_{CLM}) from Equation 13 returns to the stage (Dettmers et al. 2023). If the model outputs a malformed JSON, the L_{CLM} calculates the mathematical error between the model's output and the true desired output.

The optimizer then uses *backpropagation* to send this error signal backward through the network to update the weights. As visualized by the localized gradient annotations in Figure 9, the key efficiency of QLoRA appears during this backward pass (Dettmers et al. 2023). When the gradients (the update instructions, ∇_{W_0}) reach the massive W_0 matrix, they are simply discarded, because W_0 is frozen (Dettmers et al. 2023).

Instead, the error signal flows down the new adapter pathway via the Chain Rule of calculus. Let L represent the loss and $\frac{\partial L}{\partial h}$ represent the error signal arriving from the network's output. Because the forward pass computes $B(Ax)$, the backward pass encounters matrix B first. As shown in Equation 16, calculating the gradient for B (∇_B) relies on the cached forward outputs of A. To calculate the gradient for A (∇_A), the error signal must flow backward through the transpose of B (B^T):

$$\begin{aligned} \nabla_B &= \frac{\alpha}{r} \frac{\partial L}{\partial h} (Ax)^T \\ \nabla_A &= \frac{\alpha}{r} B^T \frac{\partial L}{\partial h} x^T \end{aligned} \tag{16}$$

This mathematical sequence shows why ∇_B can be calculated immediately using the forward activations, while calculating ∇_A explicitly requires the error signal to pass through B^T.

Understanding this localized gradient flow is critical to the feasibility of this thesis. In standard RL, agents are relatively small neural networks. However, if our training loop had to calculate and store the gradient ∇_{W_0} for all 7 billion parameters of the Qwen LLM, the workstation would immediately exceed its memory capacity. By mathematically restricting the backpropagation updates exclusively to ∇_A and ∇_B, we avoid allocating these large gradient tensors. The optimizer selectively applies these lightweight updates *only* to the low-dimensional adapter matrices (Dettmers et al. 2023).

Combined, this targeted gradient flow and aggressive 4-bit quantization reduce overall VRAM consumption by up to 75% compared to full fine-tuning (Dettmers et al. 2023). This computational efficiency is what enables our Qwen Commander to learn its JSON-formatting rules locally, running alongside the heavy MAPPO environment without exhausting the workstation's memory budget.

3.3 The Qwen Architecture and Model Selection

To serve as the cognitive controller in a MARL system, the chosen foundation model must possess spatial reasoning and reliable instruction-following capabilities. Because the model must run locally alongside the synchronous MAPPO tensor rollouts, parameter size and inference latency become the primary engineering constraints. A model that generates complex semantic plans but requires 30 seconds to output a response will stall the high-frequency reinforcement learning loop.

To identify a suitable Meta-Controller, several modern open-weights architectures were empirically evaluated via the Ollama inference framework (Ollama Team 2024). Ollama was selected for its native support of quantized models, which reduces the memory footprint of large language models while maintaining acceptable inference speeds. The candidate pool included models ranging from 2 billion to 27 billion parameters, specifically `llama3.2:latest` (2B), `qwen2.5:7b`,

`llama3.1:8b`, `gemma2:9b`, and `gemma3:27b`.

During preliminary testing, models at the extremes of this parameter spectrum proved unsuitable for the synchronous training loop. Lightweight models, such as `llama3.2` (2B), offered near-instantaneous inference but struggled to reliably respect the JSON schema required by the Python backend, frequently leading to parsing crashes. Conversely, larger models like `gemma3:27b` exhibited strong logic capabilities and consistent formatting but required approximately 17 GB of VRAM even with quantization. Given the target deployment hardware (a consumer-grade NVIDIA RTX 5070 with 12 GB of VRAM), the 27B model exceeded the physical memory budget entirely, causing immediate out-of-memory errors before the MAPPO environment could even be initialized.

Among the mid-range models (`qwen2.5:7b`, `llama3.1:8b`, and `gemma2:9b`), differences in instruction following and context processing became apparent. While both LLaMA 3.1 and Gemma 2 fit within the memory envelope, they occasionally hallucinated keys or deviated from the strict JSON hierarchy required for parsing environment commands. In contrast, `qwen2.5:7b` demonstrated highly reliable capabilities for programmatic orchestration. These two requirements—strict JSON syntax adherence and the ability to solve multi-step dependency graphs—map directly to a model's coding and logical reasoning abilities. Standard industry benchmarks support our empirical observations: as shown in Table 1, Qwen 2.5 7B significantly outperforms its peers in both zero-shot coding (HumanEval), which correlates strongly with programmatic schema adherence, and multi-step logic (GSM8K), which is vital for navigating a task DAG[1] (Qwen et al. 2024).

Table 1: Comparison of zero-shot coding and multi-step reasoning performance among mid-range models.

Model	**HumanEval (%)**	**GSM8K (%)**
Qwen 2.5 7B Instruct	**84.8**	**91.6**
Llama 3.1 8B Instruct	72.6	84.5
Gemma 2 9B	40.2	68.6

Adapted from (Qwen et al. 2024)

Because the Meta-Controller must parse spatial data and output strict, unbroken JSON commands, this gap in coding and reasoning performance is a deciding factor. Ultimately, this thesis leverages the Qwen 2.5 7B architecture (Qwen et al. 2024). At roughly 4.7 GB of VRAM usage, it fits within the hardware constraints without interfering with concurrent PyTorch processes. Qwen 2.5 strikes a practical balance between speed and reasoning: it provides fast, latency-bound token generation while possessing the logical capacity required to navigate task dependency graphs (DAGs). The architecture's extensive pre-training on code and structured data translates well to the deterministic formatting needed for an RL controller. By coupling this efficient base model with QLoRA fine-tuning, the 7B parameters can be locally specialized to act as a reliable, low-latency orchestrator.

[1] HumanEval evaluates a model's ability to synthesize functional code, while GSM8K tests its capacity to solve multi-step logical problems.

3.4 Neuro-Symbolic Orchestration and Verbal Reflection

As established in Section 2.4, pure deep reinforcement learning struggles with severe exploration bottlenecks in sparse-reward environments. While the Options Framework offers a mathematical solution to temporal abstraction, designing the rules for when to trigger an Option traditionally demands extensive human engineering.

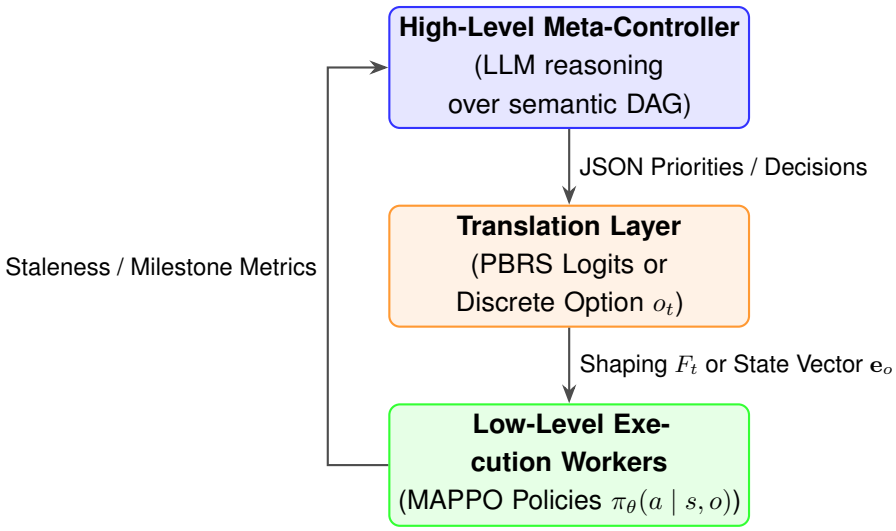
Because LLMs develop powerful logic and reasoning skills during their initial pre-training, they can be deployed as *zero-shot*[2] semantic orchestrators. To elicit these reasoning capabilities effectively, researchers developed *Chain-of-Thought* (CoT) prompting (Wei et al. 2022). By explicitly prompting the model to generate a step-by-step reasoning trace before outputting its final decision, CoT drastically improves performance on logical and multi-step tasks. Building upon this, frameworks such as ReAct (Yao et al. 2023) have demonstrated that interleaving this step-by-step language reasoning with concrete environmental actions allows models to solve complex problems autonomously. This means the model can manage tasks right out of the box, completely bypassing the need for human engineers to write manual, hard-coded decision rules.

In the specific domain of reinforcement learning, a growing body of work has emerged using LLMs as meta-level reward designers. Systems like EUREKA (Ma et al. 2024) and PRM4RL (Zhang et al. 2026) successfully deploy LLMs to automatically write reward code or generate subtask progress models to guide agent trajectories. To ensure consistent semantic progression without direct LLM prompting overhead, recent advancements like GLARE (Yan et al. 2026) successfully translate raw trajectory events into discrete temporal logic automata, mapping progress directly to structurally dense reward signals.

Rather than relying on a single pre-existing framework, this thesis synthesizes these disparate concepts into a unified, dual-tier architecture, building upon the principles of a Neuro-Symbolic Hierarchy (Mitchener et al. 2022). As illustrated in Figure 10, the high-level LLM Meta-Controller analyzes the environment and dictates strategic goals (e.g., "Craft a Pickaxe") by reasoning over a semantic Directed Acyclic Graph (DAG) of task dependencies. These text-based decisions are translated into discrete Options and passed down to the low-level MAPPO execution workers, which handle the high-frequency kinetic control (e.g., navigating the grid and avoiding obstacles).

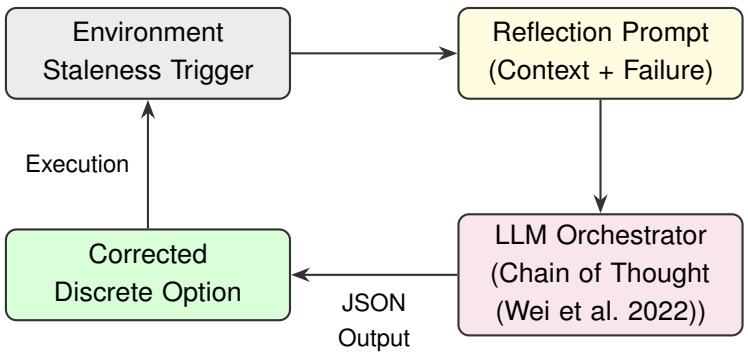
[2] In machine learning, "zero-shot" refers to a model's ability to successfully perform a task it was not explicitly trained for, relying purely on generalized knowledge acquired during pre-training.

Figure 10: The Neuro-Symbolic Hierarchy.



However, the low-level agents may sometimes fail to achieve the assigned goal. To solve this, language agents can leverage *Verbal Reflection* (Shinn et al. 2023) to evaluate and correct execution failures. As detailed in Figure 11, if the Commander issues a `MINE_IRON` option but the miner stagnates and fails to complete it after 150 steps, the environment halts. A reflection prompt containing the failure context is sent back to the LLM. This forces the model to explicitly diagnose its mistake (e.g., realizing it forgot to assign `CRAFT_PICKAXE` first) and issue a corrected macro-action.

Figure 11: The Verbal Reflection mechanism.



Adapted from (Shinn et al. 2023)

Together, these mechanisms form the neuro-symbolic feedback loop used in this thesis: the LLM handles task sequencing, and MAPPO handles spatial execution. The following chapters detail how these components are integrated, beginning with the design of the environment.

4 Environment Design and Implementation

Before detailing the neuro-symbolic architectures developed in this thesis, this chapter establishes the experimental testbed. Testing an LLM-guided MARL system requires an environment with both physical complexity (movement, spatial reasoning) and logical dependencies (tasks gated by preconditions). To this end, we engineered a custom 2D multi-agent grid-world environment from scratch.

4.1 Environment Design & Mechanics

The environment operates as a cooperative spatial grid where multiple agents must jointly gather resources, craft tools, and defeat an adversary to achieve a final objective. Unlike standard dense-reward environments, where agents receive incremental gradient signals based on Euclidean distance to a target, this environment utilizes a logical sparse-reward formulation. The agents receive extrinsic rewards solely upon the completion of discrete logical milestones (subtasks).

The overarching philosophy of this design is mathematical austerity. The environment acts as a consistent judge, providing no incremental guidance or continuous signals. It only issues a constant time penalty to encourage speed and milestone bonuses for completing predefined physical tasks.

4.1.1 Extrinsic Reward Structure

The environment reward r_t^{env} is the true, unmodified signal from the environment. It consists of a constant time-step penalty of -0.01 to encourage speed, plus milestone bonuses upon successfully completing a physical subtask:

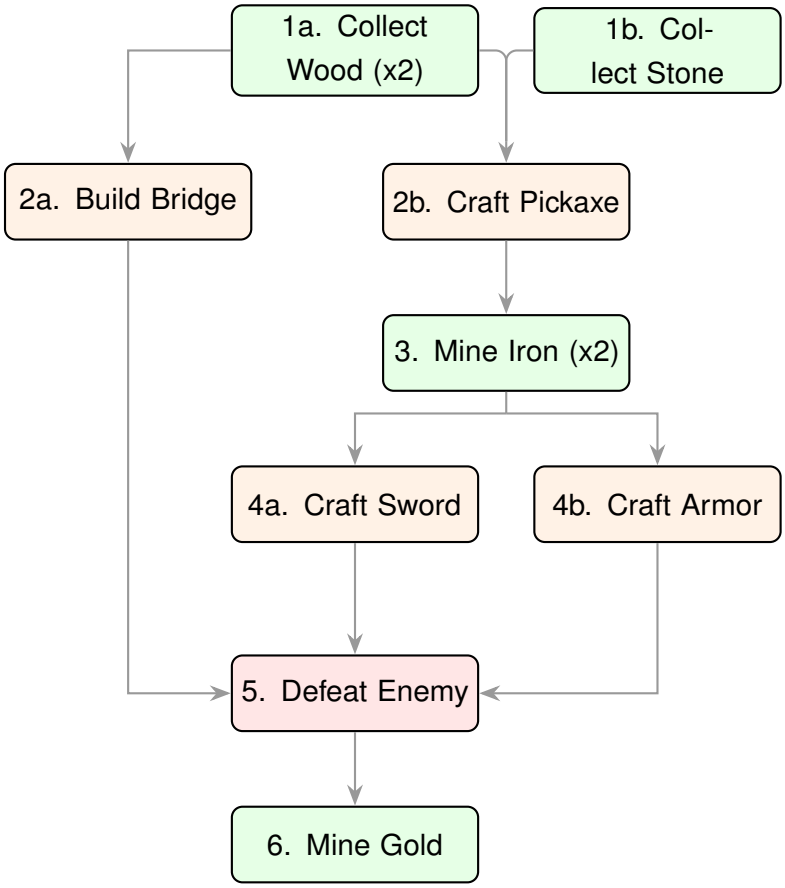
Table 2: Environment reward structure from `crafting_env.py`.

Event	Reward (shared)
Step penalty	-0.01
Collect Wood	$+2.0$
Collect Stone	$+2.0$
Craft Pickaxe	$+3.0$
Mine Iron	$+2.0$
Craft Sword/Armor	$+3.0$
Build Bridge	$+3.0$
Defeat Enemy	$+10.0$
Mine Gold (win)	$+15.0$

4.1.2 The Dependency Graph (DAG)

The fundamental exploration challenge of the environment is formulated as a Directed Acyclic Graph (DAG) of dependencies. Agents cannot arbitrarily navigate to the final goal. They must coordinate to satisfy a rigid sequence of material and infrastructural preconditions. The required topological task progression is illustrated in Figure 12.

Figure 12: The strict dependency Directed Acyclic Graph (DAG) governing environment progression. Agents must jointly acquire basic resources to clear logical bottlenecks. Notably, Wood is required for both the Pickaxe and the Bridge, while Iron supplies both Sword and Armor.



As the topological depth of this chain increases, the probability of agents discovering the complete sequence through random exploration decays. This structure isolates the exploration bottleneck that our neuro-symbolic models are designed to address.

4.1.3 Action and Observation Spaces

4.2 Partial Observability and the Action Space

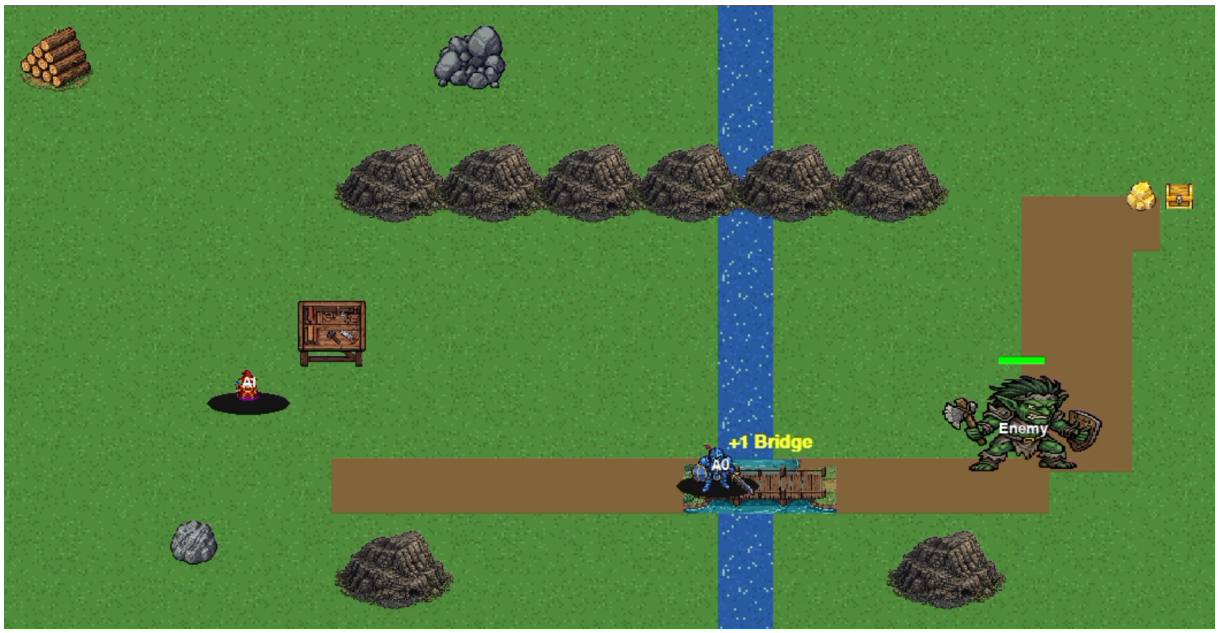
To simulate realistic constraints, the agents do not possess global vision of the grid. Instead, they operate under strict partial observability. At each time step t, agent i receives a localized observation vector $o_t^{(i)} \in \mathbb{R}^{15}$ (or $\mathbb{R}^{25}$ in Chapter 6), which includes: their own (x, y) coordinates and inventory status. The environment hides target locations (e.g., resources, workbenches) from the state vector. This requires agents to explore the spatial grid to locate objectives. To further prevent the policies from overfitting to a static sequence of movements, the agents' starting coordinates are randomized at the beginning of every episode.

The discrete action space consists of five parameters: four cardinal movement directions (Up, Down, Left, Right) and one semantic "Interact" action. The interaction mechanic is context-sensitive. For instance, triggering the action while adjacent to a forest tile chops wood, whereas executing it at a workbench (contingent upon possessing the requisite inventory) initiates tool crafting.

4.2.1 Visual Representation

Figure 13 provides a visual representation of the discrete grid-world environment utilized throughout the MARL training process. The interface renders the underlying state matrix, allowing human observers to track the spatial coordinates of the autonomous agents. While the core simulation runs headlessly to maximize throughput, this renderer serves as a tool for visual debugging and policy interpretation.

Figure 13: Visual rendering of the custom MARL environment. The interface explicitly renders the underlying state matrix, allowing human observers to track the spatial coordinates of the autonomous agents as they navigate the map.



Screenshot by Dorian THOMÉ 2026. Sprites generated by Gemini 1.5 Pro, Google. Prompt used: “generate each sprites with different state if needed like enemy alive and dead and make sure we can distinguish each elements” [generated July 10, 2026].

All visual assets utilized in this rendering engine were generated using the Gemini 1.5 Pro model via the Antigravity coding assistant platform.

4.3 Architecture & Technology Stack

Engineering a custom MARL environment demands a careful selection of the underlying technology stack, requiring a balance between visual debugging capabilities and simulation throughput.

Why Not a Game Engine? Initially, open-source game engines such as Godot (Linietsky, Manzur, et al. 2023) were considered for the simulation backbone. Godot provides a visual editor, collision physics, and a node-based hierarchy, making it an option for traditional game development.

The final constraint in our environment design is the hardware profile. Because MARL requires processing thousands of physical state transitions per second to calculate gradients, introducing an LLM query directly into the training loop introduces significant computational overhead.

As a result, the environment was engineered natively in Python, utilizing NumPy (Harris, Millman, Walt, et al. 2024) as the core mathematical engine and Pygame (Shinners et al. 2023) as

a lightweight visual wrapper. This matrix-first approach offers architectural advantages for reinforcement learning:

- **Direct Tensor Integration:** The underlying grid, entities, and inventory states exist purely as NumPy matrices. This avoids the serialization and deserialization overhead typically encountered when passing state data between a Python-based RL training script and an external C++ game engine.
- **Seamless NumPy-to-Game Translation:** Because the environment state is natively represented as a matrix, Pygame can translate these numerical coordinates into a visual interface for human debugging. The mathematical logic (kinematics, collisions) is decoupled from the rendering loop.
- **Headless Vectorization:** During the actual training loop, the Pygame visual renderer is disabled. The PyTorch models interact directly with the raw NumPy arrays, allowing a vectorized wrapper to step thousands of environments simultaneously in parallel.

4.4 Libraries & Infrastructure

The codebase relies on a modern, standardized stack of Python libraries to seamlessly interface the custom environment with the deep neural networks and the large language models:

- **MARL Interface:** The environment conforms to the PettingZoo API (Terry et al. 2021), which has established itself as the standard for multi-agent RL (analogous to the Gymnasium API for single-agent RL). Utilizing the Parallel API variant ensures the environment integrates natively with synchronous MAPPO tensor rollouts.
- **Deep Learning Framework:** PyTorch (Paszke et al. 2019) serves as the computational backbone for designing, compiling, and optimizing the MAPPO Actor and Critic networks, leveraging GPU acceleration for Generalized Advantage Estimation (GAE) and gradient descent.
- **LLM Integration:** To interface with the semantic Meta-Controllers, the Transformers library by Hugging Face is deployed (Wolf et al. 2020). For the Hierarchical Reinforcement Learning phase (which necessitates rapid, localized inference), the language models are quantized using bitsandbytes (QLoRA) (Dettmers et al. 2023) and served via the Ollama framework (Ollama Team 2024) to strictly bound the latency introduced during the training loop.
- **Experiment Tracking:** To monitor the highly stochastic training process, TensorBoard is integrated directly into the PyTorch training loop. This allows for real-time visualization of critical metrics such as the episodic return, actor-critic loss curves, and policy entropy across millions of environment transitions.

4.5 Hardware and Inference Architecture

A critical bottleneck in neuro-symbolic MARL is the severe latency introduced by the LLM during the training loop. This thesis relies on two distinct inference paradigms, online cloud APIs and localized serving, dictated by the constraints of the specific algorithms.

For the Dynamic Reward Shaping architecture, which requires infrequent, highly complex offline generation, this study leveraged the Gemini 2.5 Flash model hosted via Google AI

Studio[3]. However, integrating cloud-based models directly into the high-frequency Hierarchical Reinforcement Learning (HRL) training loop proved intractable. The MAPPO environment requires thousands of synchronous environment steps per episode. Utilizing an online model for the HRL orchestrator generated an excessive volume of requests, rapidly exhausting API rate limits and introducing network latency that stalled the GPU-bound RL rollouts.

To overcome this, the HRL training loop was executed entirely locally on a consumer-grade workstation equipped with an NVIDIA RTX 5070 GPU (12GB VRAM). Running a foundation model locally alongside the PyTorch MAPPO actor-critic tensors within a strict 12GB memory envelope necessitated specific architectural compromises. To address this, the Ollama framework (Ollama Team 2024) was deployed to provide highly optimized, low-latency local inference.

As detailed in Section 3.3, the Qwen 2.5 7B architecture was selected specifically to satisfy this strict 12GB memory envelope. When combined with 4-bit quantization via Ollama, it fits comfortably alongside the MARL experience replay buffers without displacing the PyTorch actor-critic tensors.

[3] https://aistudio.google.com/

5 Architecture 1: LLM-Driven Dynamic Reward Shaping

This chapter details the first integration strategy: LLM-driven dynamic reward shaping, informally referred to as the *Mixing Board* approach. Intuitively, this approach separates *strategy* from *execution*. The LLM acts as a Commander standing at a strategic mixing board, adjusting priority dials for various subtasks. Meanwhile, the low-level PPO agents (Yu et al. 2022) (the lumberjack and miner) act as kinetic workers, navigating the physical grid-world to fulfill whichever priority is currently dialed highest.

The Commander (our mixing board) does not act directly inside the environment. Instead, it periodically observes the agents' progress and adjusts a vector of subtask priority weights. These weights govern two things:

1. which semantic goal the agents' internal planner selects, and
2. the magnitude of the gravitational pull (potential-based shaping signal) guiding the agents toward that goal.

As established in Section 3.4, this paradigm draws foundational inspiration from recent frameworks that deploy LLMs as meta-level reward designers, such as EUREKA, PRM4RL, and GLARE (Ma et al. 2024; Zhang et al. 2026; Yan et al. 2026). While these approaches provided the theoretical justification for using LLMs to structure sparse-reward environments, the present work pivots to address the practical constraints of a high-frequency execution loop.

Rather than requiring the LLM to write complex reward functions from scratch or compile logic formulas during the rollout phase, our Commander operates at a latency-bound granularity. It simply adjusts a vector of priority weights that steer goal selection within a pre-compiled, mathematically safe potential-based shaping framework.

The exposition of this chapter follows the exact chronological dataflow of the implementation:

1. **DAG-Conditional Planning** (Section 5.2): The system prunes impossible tasks based on the agents' current inventory, preventing the Commander from issuing invalid orders.
2. **LLM Weight Query** (Section 5.4): The Commander is queried to assign raw semantic priorities to the remaining valid subtasks.
3. **Softmax Budget Normalisation** (Section 5.5): The Commander's raw priorities are mathematically smoothed into a rigorously bounded probability distribution.
4. **Potential-Based Reward Shaping** (Section 5.6): The normalized budget is injected into the training loop as a dense, magnetic reward signal to guide the workers.

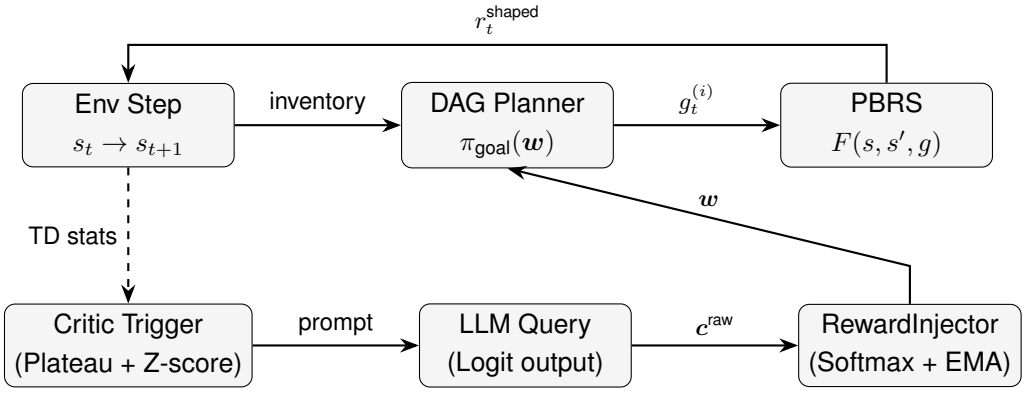
5.1 System Overview

The end-to-end pipeline of the Mixing Board architecture is illustrated in Figure 14. Conceptually, the system operates in a continuous loop. At every standard environment step, the agents update their inventory, the DAG Planner assigns them a valid target, and the PBRS module feeds them a shaped reward to guide their movement.

However, at the end of every training update, a *Critic Trigger* acts as a diagnostic monitor. It watches the agents' learning curves (TD-error and success rates). If the trigger detects that

the lumberjack and miner are stuck at a learning plateau (spinning their wheels without making progress), it halts the system. It constructs a prompt detailing the failure, queries the LLM Commander for a new strategic direction, filters that new direction through a safety layer, and updates the planner to pull the agents out of their rut.

Figure 14: Dataflow of the LLM Dynamic "Mixing Board" approach. Solid arrows denote high-frequency per-step operations (execution); the dashed arrow indicates low-frequency per-update monitoring (strategy).



5.2 DAG-Conditional Planner

Before the LLM's weights can be applied, we must ensure the Commander cannot assign impossible tasks. The crafting environment enforces a strict dependency graph over subtasks, as established in Figure 12.

5.2.1 Step-by-Step Goal Selection

To understand how the Commander's high-level strategy physically steers the low-level actors, the process can be broken down into four concrete steps:

Step 1: Filtering Feasible Tasks. Before evaluating any LLM priorities, the planner acts as a hard programmatic filter. It inspects the agents' shared inventory to build a set of *feasible* subtasks, masking out any tasks whose DAG preconditions are not met. Because LLMs are prone to hallucinations, they might occasionally assign a high priority to mining Iron even when no pickaxe exists. This step acts as an absolute constraint layer, zeroing out impossible goals to maintain mathematical stability.

Step 2 & 3: Scoring and Greedily Picking the Target. For the remaining valid tasks, the planner calculates a selection score by multiplying the hand-crafted base reward (R_j^{base}, defined in Table 2) by the LLM's dynamic priority weight (w_j). The agent then greedily selects the single subtask with the maximum score to be its active goal $g^{(i)}$:

$$g^{(i)} = \arg \max_{j \in \mathcal{F}^{(i)}(\mathbf{x})} R_j^{\text{base}} \cdot w_j \tag{17}$$

The base reward R_j^{base} provides a higher baseline value for tasks that are deeper in the dependency graph. For example, mining gold requires multiple prerequisite steps, so it naturally yields a much higher base reward than simply picking up wood. Because this "natural gravity" is

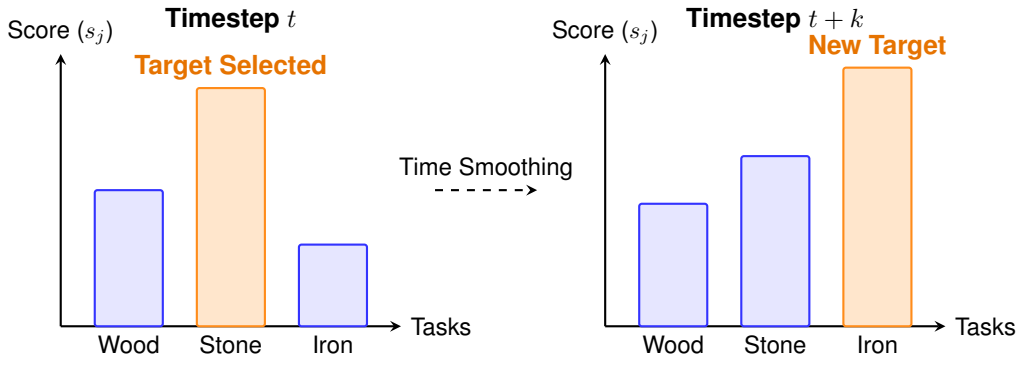
built into the base reward, the LLM only needs to gently tweak the weights w_j to shift the agent's focus, rather than generating extreme numbers.

Step 4: Cooperative Override. Because the environment is fully cooperative, certain tasks require synchronized presence. If the highest-scoring goal for either agent is a cooperative bottleneck (such as returning to the Workbench or building the Bridge), the planner overrides individual greedy selections and magnetically pulls the partner agent to the same target, ensuring they co-locate to trigger the interaction.

Finally, as visualized in Figure 15, the planner does not pass the raw scalar weight w_j to the actors. Instead, it passes the *identity* of this final chosen goal (e.g., "target: Stone") down to the Potential-Based Reward Shaping (PBRS) module. As detailed later in Section 5.6, the PBRS module acts as a GPS system: it uses that target identity to activate a specific potential function, which then provides dense, step-by-step rewards to the actors. As the actors physically navigate the grid, they receive small positive rewards for moving closer to the chosen goal, directly training their policy networks.

While the actors pursue only one goal at a time, the LLM weights for the unselected tasks are not discarded. They are smoothed over time (Section 5.5.1), meaning a lower-ranked task can steadily rise in priority until it cleanly overtakes the current leader without causing erratic behavior.

Figure 15: Visualization of Dynamic Goal Selection. The planner evaluates all feasible tasks, but only the task with the maximum score (orange) is selected as the active goal.



5.3 Two-Stage Critic Trigger

If the LLM Commander were to issue new orders at every single step, the system would collapse under inference latency, and the PPO agents would never have the stable environment necessary to learn low-level kinetic tasks. Therefore, the Commander only intervenes when the agents are genuinely stuck. The system determines this using a mathematical *two-stage gate* evaluated at the end of each PPO update iteration.

5.3.1 Stage 1: The Plateau Gate (Production Stalled)

First, the system checks if the agents have stopped making progress. Because reinforcement learning is inherently noisy, a single bad batch of trajectories does not necessarily mean the agents are stuck. Therefore, the system looks at the trend over recent time rather than overreacting to immediate fluctuations.

Specifically, it tracks the agents' success rate (ρ) for completing their assigned goals. It compares the average success rate over the *most recent* 10 updates ($\bar{\rho}_{\text{recent}}$) against the average success rate over the *previous* 10 updates ($\bar{\rho}_{\text{older}}$). The first gate opens only if the absolute improvement between these two periods falls below a negligible threshold ($\epsilon = 0.02$):

$$\text{gate\_open} \iff |\bar{\rho}_{\text{recent}} - \bar{\rho}_{\text{older}}| < \epsilon \tag{18}$$

For example, if the agents successfully mined Iron in 40% of episodes 20 steps ago, and are still hovering around 40% completion today, their learning has flatlined. This mathematical gate ensures the LLM Commander is only bothered when the lumberjack and miner have genuinely stalled. Additionally, the gate will open as a failsafe if any specific individual subtask gets permanently stuck at a partial completion rate.

5.3.2 Stage 2: The Z-Score Trigger (The Variance Trigger)

Just because production has stalled does not mean the agents need new orders. They might simply be navigating a physically long path. To confirm they are actually stuck, the system looks at the Centralized Critic.

As established in Section 2.1.3, the TD-Error (Temporal Difference Error) measures the "surprise gap" of the Critic network, i.e. the mathematical difference between its prior prediction and the newly observed reality. A high mean TD-error is not necessarily alarming. It merely indicates that the environment has a large reward scale.

However, a spike in TD-error variance is a critical failure metric. It means the Critic is fundamentally confused: it is seeing wildly different returns for the exact same states (e.g., sometimes observing a massive reward, and sometimes a penalty for identical actions). This variance spike occurs when agents get stuck at a bottleneck, thrash between uncoordinated goals, or experience severe non-stationarity (Foerster et al. 2017). Therefore, variance acts as the mathematical intervention signal of a confused agent. Once the plateau gate is open, the system checks whether the Critic's TD-error variance has spiked beyond σ standard deviations from its running mean:

$$z = \frac{\text{Var}[\delta_t] - \overline{\text{Var}[\delta]}}{\text{Std}[\text{Var}[\delta]]} > \frac{\sigma_{\text{base}}}{\max(\eta,\, 0.1)} \tag{19}$$

where δ_t is the TD error at update t, the bar denotes the running mean over a window of 100 updates, $\sigma_{\text{base}} = 1.5$ (a threshold determined empirically to balance rapid intervention against normal training noise), and η is a sensitivity factor that decays multiplicatively by 0.995 after each trigger, raising the effective threshold over time to prevent over-triggering.

If the gate is open and the Z-score trigger fires, an LLM intervention is dispatched to rescue the agents, followed by a cooldown of 25 updates to give the new orders time to take effect.

5.4 LLM Weight Query: Continuous Logit Output

When the variance trigger fires, the system constructs a structured situation report for the Commander. This prompt includes:

1. The trigger reason (plateau description, z-score value).
2. Current subtask completion percentages, noting which tasks are already fully mastered ($\geq 95\%$ success rate).
3. Per-agent TD-error statistics (mean, variance, per-agent breakdown).
4. Current reward shaping weights.
5. An episodic memory buffer of the last 3 interventions and their resulting TD-error deltas.

The prompt explicitly instructs the LLM to output unnormalized log-probabilities (logits), rather than final strict weights:

> "Your output values represent unnormalized log-probabilities (logits). They will be processed via a temperature-scaled softmax function and scaled to maintain a constant total reward budget. Focus on the RELATIVE MAGNITUDE and PRIORITY RANKING of subtasks."

The prompt explicitly requests continuous values (logits) rather than discrete binary choices (0 or 1). If the LLM acted as a binary switch, the reward landscape would shift instantaneously. In Multi-Agent RL, such rapid shifts induce severe non-stationarity that disrupts the Critic's value predictions (Foerster et al. 2017). By outputting continuous relative priorities instead, a downstream programmatic layer can mathematically blend the new values with the previous weights using an Exponential Moving Average (EMA). This programmatic smoothing ensures the agents transition gently toward their new objectives, regardless of how drastically the LLM changes its mind.

The LLM responds with a standardized JSON object containing its reasoning and the raw logits for each subtask:

```
{
  "reasoning": "<1-2 sentences>",
  "w_wood":      <float, logit>,
  "w_stone":     <float, logit>,
  "w_workbench": <float, logit>,
  "w_iron":      <float, logit>,
  "w_bridge":    <float, logit>,
  "w_enemy":     <float, logit>,
  "w_gold":      <float, logit>
}
```

By requiring the LLM to generate a brief reasoning trace before outputting the subtask weights, the system leverages Chain-of-Thought (CoT) prompting (Wei et al. 2022). This technique has been shown to significantly improve the logical consistency and interpretability of LLM decision-making.

5.5 Softmax Attention Budget

When the Commander responds to the intervention signal, the raw logits $\boldsymbol{c} = (c_1, \ldots, c_K)$ it returns are unconstrained real numbers (e.g., they can be wildly positive or negative). If we applied these raw values directly to the reward function, the Commander could accidentally inflate or deflate the entire "economy" of the grid-world, completely destabilizing the PPO Critic's value estimations.

To prevent this, the programmatic safety layer first strictly bounds the raw logits, capping any individual task logit at a maximum of 3.0 and ensuring the total sum across all tasks does not exceed 15.0. These bounded logits are then converted into a *budget-preserving* weight vector via a temperature-scaled softmax (Sutton et Barto 2018):

$$\boxed{w_i^{\text{softmax}} \;=\; K \;\cdot\; \frac{\exp\!\big(c_i/\tau\big)}{\displaystyle\sum_{j=1}^{K} \exp\!\big(c_j/\tau\big)}} \tag{20}$$

where $K = |\mathcal{W}| = 7$ is the total number of subtasks, and $\tau = 0.5$ is an empirically tuned temperature hyperparameter. This operation transforms the raw LLM outputs c into normalized weights w^{softmax} with the following properties:

- *Constant budget*: $\sum w_i^{\text{softmax}} = K$. The total reward distributed among the tasks remains perfectly constant (exactly 7.0), regardless of the LLM's output scale or the prior 15.0 bound on the raw logits.
- *Decisiveness via Temperature*: A temperature $\tau < 1.0$ mathematically decreases the entropy of the resulting distribution. Setting $\tau = 0.5$ deliberately *sharpens* the LLM's raw priorities. This amplifies the Commander's confidence, ensuring the highest-ranked task receives a distinctively larger share of the reward budget, which provides a clear and decisive gradient signal for the agents to follow.
- *Numerical stability*: To prevent overflow errors, the logits are shifted by subtracting their maximum value before exponentiation.

5.5.1 EMA Smoothing

Sudden transitions in target goals cause abrupt policy shifts for the agents. To ensure the active weights shift smoothly, the newly computed w^{softmax} is blended with the previously active weights via an Exponential Moving Average (EMA), a standard stabilization technique in RL:

$$\widetilde{w}_i^{(t)} \;=\; \alpha \cdot w_i^{\text{softmax}} + (1-\alpha) \cdot \widetilde{w}_i^{(t-1)} \tag{21}$$

with $\alpha = 0.2$. This mechanism acts as a shock absorber. Because the EMA is a convex combination, if both the target w^{softmax} and the previous weights sum to K, the new smoothed weights $\widetilde{w}^{(t)}$ are mathematically guaranteed to also sum to exactly K, perfectly preserving the budget constraint over time. Furthermore, it ensures that each new directive from the Commander takes roughly 5 updates to fully propagate into the environment, giving the agents time to adjust their trajectories.

5.5.2 Auto-Rollback

Despite these mathematical safeguards, the system employs a final safety net inspired by Safe Reinforcement Learning (Amodei et al. 2016). The safety layer monitors the average environment reward for 25 updates following any LLM intervention. If this post-intervention average drops more than 20% (an empirically chosen safety threshold) below the pre-intervention level, it triggers an *Auto-Rollback*:

$$\text{rollback} \iff \frac{\bar{r}_{\text{pre}} - \bar{r}_{\text{post}}}{|\bar{r}_{\text{pre}}|} > 0.20 \tag{22}$$

Fundamental policy optimization algorithms, including PPO, are designed around the mathematical desire for monotonic improvement, guaranteeing that a new policy performs at least as well as the old one (Schulman et al. 2017). Because the LLM injects external, non-stationary shocks into the reward landscape, these mathematical guarantees are temporarily broken. The Auto-Rollback acts as an empirical failsafe, instantly reverting the weights to restore training stability if the Commander's new strategy proves disastrous on the ground.

It should be explicitly noted that while this Auto-Rollback mechanism was fully implemented as part of the system's theoretical architecture, it was deliberately deactivated during the experimental evaluations discussed in Chapter 7. This decision was made to observe the raw, unmitigated impact of the LLM interventions, both positive and negative, on the learning dynamics, ensuring that the failsafe did not artificially mask the consequences of poor strategic suggestions.

5.6 Potential-Based Reward Shaping

Once the planner evaluates the smoothed weights and assigns an active goal zone $g_t^{(i)}$ to agent i, the environment must translate that goal into a physical force. This is achieved via the potential-based reward shaping (PBRS) framework established in Section 2.4.1.

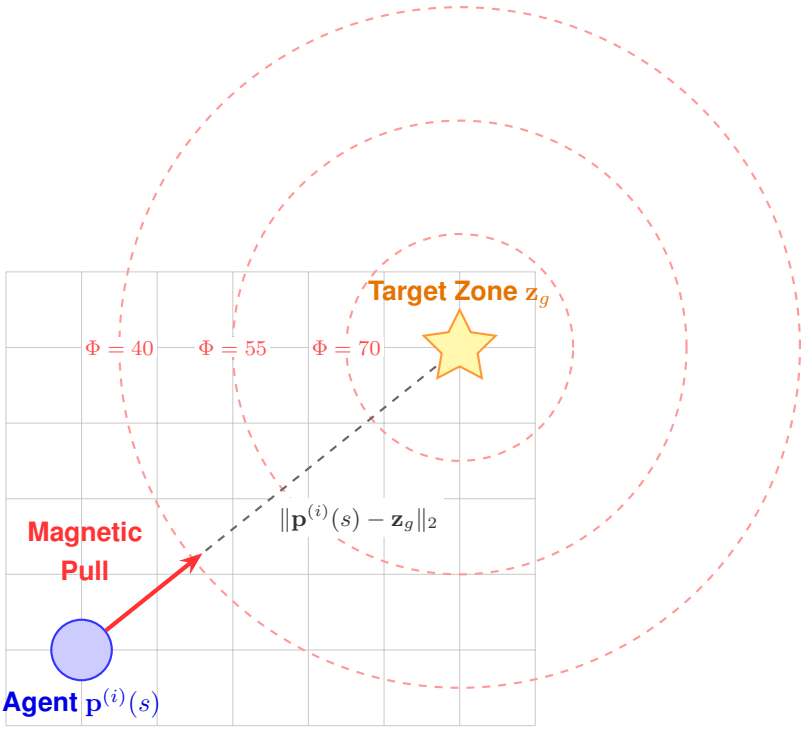
5.6.1 The Magnetic Pull (Potential Function)

As introduced theoretically in Chapter 2.2, Potential-Based Reward Shaping relies on a state-based potential function $\Phi(s)$. In this specific grid-world implementation, the potential of agent i with respect to its assigned goal zone g is calculated geometrically:

$$\Phi^{(i)}(s, g) = D_{\max} - \|\mathbf{p}^{(i)}(s) - \mathbf{z}_g\|_2 \tag{23}$$

where $\mathbf{p}^{(i)}(s) \in \mathbb{R}^2$ is the agent's (x, y) position, $\mathbf{z}_g \in \mathbb{R}^2$ is the target zone's position, and $D_{\max} = 85$ is the maximum diagonal distance of the grid. This provides a strict scalar measure of proximity: it is high when the agent is physically close to the goal and low when it is far away.

Figure 16: Visualizing the Potential Function Φ. The scalar potential strictly increases as the agent approaches the active target, creating a directional pull that guides navigation.



5.6.2 Applying the Shaping Reward

As the agent steps from state t to $t+1$, the per-step shaping reward measures the difference in this potential:

$$F_t^{(i)} \;=\; \left(\gamma\,\Phi^{(i)}(s_{t+1}, g) - \Phi^{(i)}(s_t, g)\right) \;\cdot\; \lambda_{\text{scale}} \;\cdot\; \mathbb{1}[g \text{ active}] \tag{24}$$

As established in Section 2.4.1, $\gamma = 0.99$ acts as the anti-farming mechanism, mathematically punishing the agent if it loops backward along its own path.

The scale factor ($\lambda_{\text{scale}} = 0.05$) acts as a volume knob. If an agent runs a long distance across the map, its potential Φ might jump by $+20$ points. If unscaled, the agent would receive $+20$ simply for running, making continuous movement more lucrative than actually mining Gold (which only yields a true sparse reward of $+15$). Multiplying by 0.05 shrinks that $+20$ movement reward down to a $+1.0$ nudge. This guarantees the dense shaping reward remains a gentle guide rather than an exploitable feedback loop, while the sparse environmental milestones remain the ultimate prize.

Additionally, the indicator $\mathbb{1}[g \text{ active}]$ acts as a mask. If the agent completes its portion of the DAG and is idling while waiting for its partner, this mask zeroes out the shaping reward entirely, preventing the agent from being incorrectly penalized for remaining stationary.

To execute this geometric logic simultaneously across 128 parallel environments efficiently, the scaling relies entirely on vectorized tensor operations, bypassing Python loops.

The Commander's influence is encapsulated entirely within the goal selection variable g (Equation 17). The LLM dictates *which* direction the magnetic pull originates from, but it does not change the *strength* of that pull. If F were dynamically scaled by the shifting LLM weights

w_j, the underlying physical laws of the grid-world would constantly change, creating massive non-stationarity that would crash the PPO Critic. By isolating the strategy (the target g) from the force (the stable, policy-invariant gradient F), the agents can learn to navigate safely.

5.6.3 Mathematical Impact on the MAPPO Objective

In this Mixing Board architecture, the LLM does not alter the Actor network's state observations; the agents remain completely blind to the LLM's existence. Instead, the LLM alters the environment's reward signal r_t by injecting the shaping reward $F_t^{(i)}$ (scaled by a time-varying decay factor λ_t):

$$r_t^{\text{total},(i)} = r_t^{\text{env}} + \lambda_t F_t^{(i)} \tag{25}$$

This augmented reward flows directly into the MAPPO Advantage calculation $A_t^{(i)} = r_t^{\text{total},(i)} + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$. As a result, the Centralized Critic network $V_\phi(s_t)$ adjusts its value estimations to predict this newly dense, goal-oriented landscape, and the Actor simply follows the updated Advantage gradient.

To ensure the agents eventually learn to act independently, the blending coefficient λ_t from Equation 25 forces the shaping signal to fade out over time. The decay λ_t begins at 1.0 and anneals linearly to 0.01 over 500 updates, but only *after* the agents achieve a 40% success rate on the critical "Bridge" bottleneck. This schedule ensures the agents have dense "training wheels" early in the learning process, and gracefully transitions them to the sparse environment reality once they prove they understand the map's topology.

5.7 Async Communication Architecture

In standard Markov Decision Processes, observation and action delays severely destabilize training (Katsikopoulos et Engelbrecht 2003). If the PPO workers paused to wait for the LLM Commander's response during every intervention trigger, network latency and inference time would bottleneck the simulation, dropping GPU utilization to near zero.

To prevent this, queries are dispatched asynchronously via a daemon thread. This approach mirrors standard asynchronous reinforcement learning architectures (e.g., Petrenko et al. 2020), decoupling the high-frequency environment clock of the PPO workers from the variable latency of the LLM inference pipeline. The dataflow executes as follows:

1. **Trigger:** The critic trigger fires at the end of a PPO update and constructs the situation report prompt.
2. **Dispatch:** The prompt is enqueued to a thread-safe queue.
3. **Background Processing:** The daemon thread queries the LLM API without interrupting the main simulation loop.
4. **Continuous Execution:** While the LLM generates a response, the RL agents continue interacting with the environment using the *previous* priority weights.
5. **Seamless Injection:** Once inference completes, the new weights are stored as "pending." At the next PPO update boundary, the safety layer applies EMA smoothing and seamlessly updates the active shaping targets.

Ultimately, this asynchronous delay does not destabilize the agents because the LLM only shifts

the *macro-strategy* (the target zone), rather than controlling step-by-step micro-actions. To further ensure system robustness, the asynchronous bridge employs a thread-safe cache to eliminate redundant queries, and an exponential backoff mechanism (2^k seconds, up to 3 retries) to safely handle API timeouts, allowing identical prompts to instantly recycle previous LLM outputs.

5.8 Summary

The Mixing Board architecture utilizes the LLM as a *meta-level Commander* to parse aggregate training statistics and output strategic priority logits. To prevent training instability, these priorities are routed through strict programmatic safety layers: DAG guardrails, bounding clips, softmax budget normalization, EMA smoothing, and an auto-rollback failsafe. Finally, the Potential-Based Reward Shaping (PBRS) framework translates this strategic intent into a dense, policy-invariant directional pull, successfully guiding the MAPPO workers out of exploration bottlenecks while preserving the mathematical convergence guarantees of the underlying RL algorithm.

Table 3: Key hyperparameters of the Mixing Board approach.

Component	**Parameter**	**Value**
Softmax budget	Temperature τ	0.5
	Budget size K	7
EMA smoothing	α	0.2
Safety bounds	$w_{\max}$ (per-weight clip)	3.0
	W_{bound} (sum clip)	15.0
Critic trigger	Plateau window W	10
	Plateau threshold ϵ	0.02
	Z-score σ_{base}	1.5
	Cooldown	25 updates
	Sensitivity decay	0.995
Auto-rollback	Window	25 updates
	Drop threshold	20%
PBRS	Scale λ_{scale}	0.05
	$D_{\max}$	85.0
Shaping decay	λ_{init}	1.0
	λ_{final}	0.01
	Decay trigger	bridge $>$ 40%
	Decay duration	500 updates

6 Architecture 2: Hierarchical RL with an LLM Meta-Controller

The previous chapter established the *Mixing Board* architecture, representing a "soft integration" of Large Language Models into MARL. In that paradigm, the LLM acted as an environmental architect, manipulating the gravitational pull of potential-based rewards to gently guide the agents toward logical milestones. While this separation preserves the underlying Markov properties, it restricts the LLM to an indirect role: it can only suggest a direction, not enforce a strategy.

To fully explore the spectrum of neuro-symbolic integration, we must ask: what happens when we grant the LLM direct semantic authority over the agents' cognitive state?

This section details the second integration strategy: a Hierarchical Reinforcement Learning (HRL) framework. Here, we transition from soft environmental guidance to hard architectural integration. The LLM ceases to be a distant architect and becomes a direct *Meta-Controller*. Instead of adjusting continuous weights over a fixed greedy planner, the Meta-Controller replaces the planner entirely. It selects discrete high-level *Options* (macro-actions) that strictly govern the behavior of the low-level MAPPO policy.

In our established metaphor, the Commander is no longer just shifting the gravity; it is issuing explicit, discrete commands directly to the sensory inputs of the lumberjack and miner.

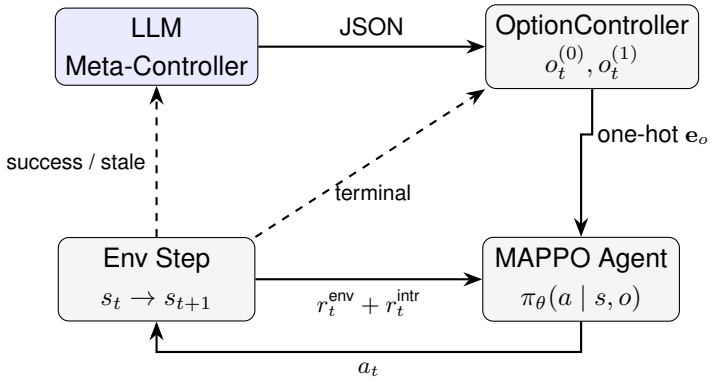
The exposition follows the implementation order:

1. **Option Definitions** (section 6.2): Defining the discrete macro-actions available to the Commander.
2. **Discrete Option Selection** (section 6.3): How the LLM is triggered to assign these explicit tasks.
3. **One-Hot Observation Augmentation** (section 6.4): How the Commander's text-based orders are mathematically injected into the MAPPO neural network.
4. **Bipartite Reward Structure** (section 6.5): The dual-reward system required to train the workers to follow these new explicit orders.
5. **Staleness and Counterfactual Reflection** (section 6.7): How the Commander diagnoses its own mistakes when an order fails on the ground.

6.1 System Overview

Figure 17 illustrates this two-level command architecture. The Meta-Controller (LLM) observes inventory transitions and assigns explicit Options to each agent asynchronously. Unlike the previous approach where the MAPPO policy remained blind to the LLM's existence, here, the low-level MAPPO policy receives the active option directly as part of its mathematical observation space. This creates a true, tightly coupled two-level decision hierarchy.

Figure 17: Two-level HRL architecture. The LLM Meta-Controller is triggered by option success or staleness (dashed). The OptionController passes one-hot embeddings to the MAPPO agent at every step (solid), directly altering the agent's cognitive state.



6.2 Option Definitions

In the dynamic shaping approach, the LLM merely influenced a hard-coded planner. In this Hierarchical Reinforcement Learning (HRL) architecture, the LLM completely replaces the planner. It selects from a finite set of explicit macro-actions, or *Options*, and directly commands the low-level MAPPO agents.

An Option $o \in \mathcal{O}$ is mathematically defined by the triple $\langle \mathcal{I}_o, \pi_o, \beta_o \rangle$: an initiation set, an intra-option policy, and a termination condition (Sutton, Precup et Singh 1999). In our framework, the option set $\mathcal{O}$ is derived directly from the dependency DAG, providing 10 discrete macro-actions as detailed in Table 4.

Table 4: Option properties: target zone, responsible agent, and initiation condition.

Option	**Zone**	**Agent**	**Initiation Condition**
`COLLECT_WOOD`	Wood (0)	0	Always
`COLLECT_STONE`	Stone (1)	1	Always
`CRAFT_PICKAXE`	Workbench (2)	Both	wood ≥ 1, stone ≥ 1, pickaxe < 1
`MINE_IRON`	Iron (3)	1	pickaxe ≥ 1
`CRAFT_SWORD`	Workbench (2)	Both	iron ≥ 1, sword < 1
`CRAFT_ARMOR`	Workbench (2)	Both	iron ≥ 1, armor < 1
`BUILD_BRIDGE`	Bridge (4)	Both	wood ≥ 1, bridge < 1
`FIGHT_ENEMY`	Enemy (5)	Both	bridge, sword, armor ≥ 1
`COLLECT_GOLD`	Gold (6)	1	enemy defeated, gold < 1
`IDLE`	—	Both	Always

Rather than building 10 separate neural networks, the intra-option policy π_o is executed by the single, shared MAPPO network, which conditions its kinetic movements based on the active option (detailed in Section 6.4).

The termination condition (β_o) is not based on spatial coordinates, but on semantic success. For example, if the Commander radios the `COLLECT_WOOD` Option to the lumberjack, that Option only terminates when the system registers an increase in the wood inventory.

6.3 Discrete Option Selection by the LLM

Because the Meta-Controller issues explicit, discrete commands, it does not need to continuously monitor the mathematical variance of the Critic network (unlike the Mixing Board approach). Instead, the Commander acts reactively, stepping in only when a worker completes a task or fails to make progress.

6.3.1 Trigger Conditions: Success and Staleness

Option selection is triggered dynamically under two specific conditions, evaluated per-environment:

1. **Success:** The agent successfully acquired the item (the inventory slot changed), meaning the current Option has terminated and the agent awaits a new order.
2. **Staleness:** The active option has been running for more than $T_{\text{stale}} = 150$ steps without completion. In MARL, a step $t \to t{+}1$ represents a physical movement or action in the grid. If an agent executes 150 low-level kinetic actions without satisfying its macro-objective, the system assumes the agent is stuck (e.g., trying to mine iron without a pickaxe) and forces an LLM intervention.

Environments that are already awaiting an LLM response (`llm_pending`) or are on a brief 50-step cooldown are safely excluded from triggering new queries.

6.3.2 Prompt Structure

When a trigger fires, the system translates the grid-world's physical inventory state into a text-based JSON schema. For a standard *Success* trigger, the Commander receives a Chain-of-Thought prompt (Wei et al. 2022) detailing the current inventory and the DAG dependencies, requesting exactly one discrete Option per agent:

```
You are the high-level Cognitive Orchestrator for two MARL agents.
Strict Dependency DAG: (Wood+Stone) -> Pickaxe -> Iron
    -> (Sword+Armor) -> Bridge -> Enemy -> Gold.

### CURRENT STATE:
Inventory: {inventory}
Agent 0 Status: {a0_status}
Agent 1 Status: {a1_status}

### YOUR TASK:
Assign exactly ONE discrete Option to each agent.
Available Options: ["COLLECT_WOOD", "COLLECT_STONE", "CRAFT_PICKAXE",
    "MINE_IRON", "CRAFT_SWORD", "CRAFT_ARMOR", "BUILD_BRIDGE",
    "FIGHT_ENEMY", "COLLECT_GOLD", "IDLE"]

Respond ONLY with valid JSON:
{
  "dag_check": "<1 sentence reasoning>",
  "agent_0_option": "<Option>",
  "agent_1_option": "<Option>"
}
```

6.3.3 Asynchronous Query Management

Training MARL across 128 parallel environments generates significant query volume. To optimize throughput, the system deduplicates queries by grouping environments with identical inventory states, sending a single prompt per unique state to the LLM, and broadcasting the response to all matching environments.

Additionally, because LLM inference takes several seconds, continuous environment execution would cause **state drift** (latency-induced non-stationarity) (Katsikopoulos et Engelbrecht 2003), rendering the LLM's eventual decision outdated. Following established LLM-agent practices (Zhu et al. 2023), any environment awaiting a response is paused. This ensures the Meta-Controller's decision is applied to the exact state it observed, maintaining strict temporal alignment between strategy and execution.

6.4 One-Hot Observation Augmentation

Once the Commander has selected a discrete Option, that command must be physically transmitted to the worker. In the MAPPO framework, this is achieved by appending a one-hot vector directly into the agent's localized observation space, effectively acting as a "radio receiver" for the Commander's orders. We can see how exactly in Equation 26.

6.4.1 Dynamic Enemy State

Unlike the Mixing Board approach, which utilized the baseline static environment, the HRL architecture operates in an extended environment that introduces a *dynamic enemy*. In this variant, the enemy is no longer a static zone interaction. Instead, the enemy tracks the agents' positions and actively chases any agent that crosses the river, creating a real-time pursuit challenge. The combat system is designed to reward coordination: if both agents attack the enemy simultaneously, they deal 50 HP of damage per step, compared to only 10 HP for a solo attack. While this $5\times$ coordination multiplier was intended to make the `FIGHT_ENEMY` option inherently cooperative, the overarching time penalty (-0.01 per step) often creates conflicting incentives. As explored in Chapter 7, the agents occasionally discover that having one agent rush the enemy while the other crafts is more time-efficient than waiting to launch a coordinated assault.

To give the agents spatial awareness of this dynamic threat, the three dimensions that were occupied by the goal embedding in the Mixing Board approach (indices 2–4) are repurposed to carry the enemy's real-time state: its (x, y) position and its normalized health $h/100$. Providing this state information is standard practice in fully observable Markov Decision Processes (MDPs); it simply represents the agents' physical line-of-sight, which is required for the low-level PPO policy to learn the kinetic micro-actions of pursuit and combat.

6.4.2 Observation Construction

The vector input to the MAPPO network in this HRL mode is expanded to a 25-dimensional vector:

$$\mathbf{v}_t^{(i)} = \left[\underbrace{\mathbf{p}^{(i)}}_{\mathbb{R}^2} \,\middle\|\, \underbrace{\mathbf{d}^{\text{enemy}}}_{\mathbb{R}^3} \,\middle\|\, \underbrace{\mathbf{x}}_{\mathbb{R}^{10}\text{ (inv)}} \,\middle\|\, \underbrace{\mathbf{e}_{o_t^{(i)}}}_{\mathbb{R}^{10}\text{ (one-hot)}} \right] \in \mathbb{R}^{25} \tag{26}$$

where $\mathbf{p}^{(i)}$ is the agent's spatial position, $\mathbf{d}^{\text{enemy}} = (x_{\text{enemy}}, y_{\text{enemy}}, h_{\text{enemy}}/100)$ is the dynamic enemy state vector, $\mathbf{x}$ is the shared inventory vector, and $\mathbf{e}_o \in \{0,1\}^{10}$ is the one-hot encoding of the active option.

For example, if an agent is located at coordinates $(3, 4)$, the enemy is at $(5, 5)$ with 50% health, the inventory contains exactly 1 wood, and the Commander has assigned the first option (e.g., `COLLECT_WOOD`, index 0), the augmented vector is constructed as follows:

$$\mathbf{v}_t^{(i)} = \left[\underbrace{3,\ 4}_{\mathbf{p}^{(i)}} \,\middle\|\, \underbrace{5,\ 5,\ 0.5}_{\mathbf{d}^{\text{enemy}}} \,\middle\|\, \underbrace{1,\ 0, \ldots, 0}_{\mathbf{x} \in \mathbb{R}^{10}} \,\middle\|\, \underbrace{1,\ 0, \ldots, 0}_{\mathbf{e}_o \in \mathbb{R}^{10}} \right] \tag{27}$$

This design expands the $\mathbb{R}^{15}$ observation space of the Mixing Board approach to $\mathbb{R}^{25}$. Rather than increasing the spatial observation dimensionality, the three dimensions previously occupied by the continuous goal embedding are efficiently repurposed to carry the enemy state. The additional 10 dimensions are strictly dedicated to the one-hot option vector.

By appending this one-hot vector $\mathbf{e}_{o_t^{(i)}}$ directly to the observation space, the MAPPO network learns an *option-conditional policy* $\pi_\theta(a \mid s, o)$. This one-hot encoding acts as a mathematical switch, allowing a single neural network to dynamically shift its behavior based on the Commander's active assignment, avoiding the need to train a separate policy network for each of the 10 distinct options.

6.5 Bipartite Reward Structure

In standard RL, the environment reward is the ground truth. However, in our sparse environment, relying solely on this is analogous to receiving feedback only at the very end of a long episode. To ensure the agents learn the day-to-day behaviors required to achieve that final milestone, the HRL approach uses a bipartite composite reward: a sparse environment reward and a dense intrinsic reward.

6.5.1 Extrinsic and Intrinsic Components

As established in Section 4.1, the extrinsic environment reward r_t^{env} is the true, unmodified sparse signal from the grid-world (refer to Table 2). It consists of a constant time-step penalty of -0.01 to encourage speed, plus massive milestone bonuses upon successfully completing a physical subtask.

Conversely, the intrinsic (shaping) reward is a dense synthetic shaping signal generated internally by the architecture to keep the agent motivated toward the Commander's assigned target. It consists of two sub-components: a continuous distance-based shaping mechanism, and a

discrete option success bonus.

First, the continuous distance-based shaping reward toward the active option's target zone is computed identically to the PBRS framework (Ng, Harada et Russell 1999) established in Section 2.4.1, but applied to the option's target rather than the planner's target:

$$r_t^{\text{shape},(i)} \;=\; \mathbb{1}_{\{o^{(i)}\neq \texttt{IDLE}\}} \;\cdot\; \lambda_{\text{intr}}\big(\gamma\,\Phi^{(i)}(s_{t+1}, z_o) - \Phi^{(i)}(s_t, z_o)\big) \tag{28}$$

where z_o is the target zone of option o (table 4), Φ is the potential function, and $\lambda_{\text{intr}} = 0.05$ is the shaping scaling factor (equivalent to λ_{scale} from the Mixing Board approach).

As established in previous chapters, this canonical difference form $\gamma\,\Phi(s_{t+1}) - \Phi(s_t)$ guarantees policy invariance and naturally prevents the agents from exploiting the reward signal by stepping back and forth (Ng, Harada et Russell 1999).

The indicator function $\mathbb{1}_{\{o^{(i)}\neq \texttt{IDLE}\}}$ acts as a coordination mask. An agent assigned the `IDLE` option is actively waiting for its partner to satisfy a dependency; applying distance-based shaping here would incorrectly penalize it for remaining stationary. The mask zeroes out this shaping signal, allowing the agent to safely wait without destabilizing the value network.

Second, when an agent completes its active option (the relevant inventory slot increases), the distance shaping is overridden, and it receives a flat intrinsic success bonus of $+1.0$:

$$r_t^{\text{intr},(i)} = \begin{cases} 1.0 & \text{if option } o^{(i)} \text{ terminated successfully} \\ r_t^{\text{shape},(i)} \text{ (eq. (28))} & \text{otherwise} \end{cases} \tag{29}$$

Success is detected by comparing the inventory before and after the environment step.

6.5.2 Composite Reward

The total reward signal fed to the PPO algorithm is a weighted sum of the environment diploma and the intrinsic breadcrumbs:

$$\boxed{r_t^{\text{total}} = r_t^{\text{env}} + \alpha_{\text{intr}} \cdot r_t^{\text{intr}}} \tag{30}$$

with $\alpha_{\text{intr}} = 0.5$. This coefficient was chosen to balance the dense intrinsic signal against the sparse environmental milestones; too large a value causes the policy to over-optimise for intrinsic shaping at the expense of actual task completion.

Note the key difference from the Mixing Board approach: here, no λ decay schedule is applied. The intrinsic coefficient remains fixed because the LLM continuously updates the *direction* (which option to pursue) rather than the magnitude of shaping.

6.6 Mathematical Impact on the MAPPO Objective

Unlike the Dynamic approach, which acted purely as an environment architect through the reward signal, the Meta-Controller strikes the MAPPO equations in *two places simultaneously*: the observation space (Actor) and the reward space (Critic).

On the Actor side, the one-hot option conditioning transforms the decentralized policy of agent i from $\pi_\theta(a_t^{(i)}|s_t^{(i)})$ into an option-conditional distribution $\pi_\theta(a_t^{(i)}|s_t^{(i)}, o_t^{(i)})$. The Actor's goal in PPO is to maximize the probability of good actions, but it uses a "clipped objective" to ensure the network does not update its weights too aggressively in a single step. By injecting the discrete option $o_t^{(i)}$, the probability ratio $r_t^{(i)}(\theta)$ between the new and old policy becomes option-aware:

$$r_t^{(i)}(\theta) = \frac{\pi_\theta(a_t^{(i)}|s_t^{(i)}, o_t^{(i)})}{\pi_{\theta_{\text{old}}}(a_t^{(i)}|s_t^{(i)}, o_t^{(i)})} \tag{31}$$

and the clipped surrogate objective updates accordingly:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}\left[\min\left(r_t^{(i)}(\theta)A_t^{(i)},\ \text{clip}(r_t^{(i)}(\theta), 1-\epsilon, 1+\epsilon)A_t^{(i)}\right)\right] \tag{32}$$

On the Critic side, learning is driven by the Advantage function ($A_t^{(i)}$). Remember that the Advantage measures how much better a chosen action was compared to the baseline expectation. This baseline is provided by the Centralized Value function V_ϕ. To maintain the CTDE paradigm established in Section 2.2.2, the Critic evaluates the *global* environment state $\mathbf{s}_t$. In this HRL architecture, it also receives the joint options of all agents $\mathbf{o}_t = (o_t^{(0)}, \ldots, o_t^{(N)})$. The bipartite reward feeds into this option-conditional Advantage:

$$A_t^{(i)} = r_t^{\text{total},(i)} + \gamma V_\phi(\mathbf{s}_{t+1}, \mathbf{o}_{t+1}) - V_\phi(\mathbf{s}_t, \mathbf{o}_t) \tag{33}$$

Because the Value function V_ϕ now receives the globally augmented state $\mathbf{s}'_t = [\mathbf{s}_t \parallel \mathbf{e}_{\mathbf{o}_t}]$, the Critic learns to predict entirely different baseline expected returns depending on the combination of options actively guiding the agents.

The consequence is a tightly coupled learning loop: the Actor gradient explicitly links the local physical state $s_t^{(i)}$, the Commander's specific order $o_t^{(i)}$, and the intrinsic success of that command through $A_t^{(i)}$. The network learns distinct sub-policies for each option, not through separate networks, but through a single policy that conditions its behavior on which option the LLM has assigned.

6.7 Execution Lifecycle and Stability Guardrails

To ensure the HRL architecture remains robust over millions of training steps, several lifecycle management and stability mechanisms are integrated directly into the environment loop.

6.7.1 Episode Initialization and Reset

When an environment reaches a terminal state (gold mined, game over, or truncation at 800 steps), the environment is hard-reset. The agents are assigned initial options corresponding to the first layer of the DAG (`COLLECT_WOOD` for the lumberjack, `COLLECT_STONE` for the miner), ensuring they begin collecting resources immediately upon episode start without waiting for an initial LLM query.

6.7.2 Staleness Detection and Counterfactual Reflection

During execution, an internal watchdog tracks the age (number of steps) of each active option. If an option exceeds $T_{\text{stale}} = 150$ steps without termination, the environment is marked as stale and the LLM is re-queried. The `IDLE` options are excluded from this trigger, as intentional waiting is a valid cooperative strategy.

When a staleness trigger fires, the LLM receives a *counterfactual reflection prompt* instead of the standard assignment prompt. Rather than simply asking for a new assignment, this prompt forces the LLM to diagnose **why** its previous assignment failed (e.g., a missing DAG precondition like `MINE_IRON` without a pickaxe) before issuing new options:

```
### REFLECTION REQUIRED:
Agent 0 was assigned "{a0_option}" and Agent 1 was assigned "{a1_option}".
After {stale_steps} steps, NEITHER has succeeded.

### YOUR TASK:
1. First, diagnose what prerequisite is MISSING.
2. Then assign corrected options that respect the DAG.

Respond ONLY with JSON:
{
  "reflection": "<diagnosis>",
  "dag_check": "<verification>",
  "agent_0_option": "<Option>",
  "agent_1_option": "<Option>"
}
```

This mechanism leverages the LLM's zero-shot reasoning to debug its own prior hallucinations, synergizing reasoning with counterfactual critique (Yao et al. 2023; Shinn et al. 2023).

6.7.3 Target KL Divergence Guard

To protect the policy network from destructive updates when intrinsic reward signals shift abruptly after an LLM reassignment, the training loop employs a target KL divergence early-stopping mechanism (Schulman et al. 2017). During each PPO epoch, the approximate KL divergence between the old and new policies is monitored:

$$D_{\text{KL}}^{\text{approx}} = \mathbb{E}[-\log \pi_{\theta_{\text{new}}}(a \mid s, o) + \log \pi_{\theta_{\text{old}}}(a \mid s, o)] \tag{34}$$

If $D_{\text{KL}}^{\text{approx}} > 0.015$ (an empirical threshold widely adopted in standard PPO implementations), the remaining PPO epochs are aborted for that update, ensuring stable policy convergence.

6.8 Comparison with the Mixing Board Approach

Before detailing the specific fine-tuning pipeline required for the Meta-Controller, it is instructive to contrast this architecture with the Dynamic Reward Shaping (Mixing Board) approach developed in Chapter 5. While both paradigms seek to integrate LLM reasoning into MARL, they operate at fundamentally different levels of abstraction.

The Mixing Board is a *soft integration* strategy: the LLM gently nudges the priority of intrinsic rewards, but the MARL agents remain entirely responsible for discovering the physical behaviors. Conversely, the Meta-Controller represents a *hard integration*: the LLM acts as a strict supervisor, dictating explicit macro-actions (options) that the agents must execute. This shift from continuous priority logits to discrete option assignments necessitates significant structural changes, particularly regarding LLM query frequency, observation augmentation, and safety mechanisms. These core architectural differences are summarized in Table 5.

Table 5: Structural comparison of the two LLM integration strategies.

Aspect	**Mixing Board (Soft Integration)**	**Meta-Controller (Hard Integration)**
LLM output space	Continuous logits $\in \mathbb{R}^7$	Discrete options $\in \mathcal{O}^2$
LLM controls	Subtask priority weights	Explicit per-agent macro-actions
Trigger mechanism	Plateau + Z-score on TD error	Option success or staleness
Query granularity	Global (once per trigger)	Per-environment (batched)
Observation augmentation	None (blind to LLM)	One-hot option $\in \{0,1\}^{10}$
Reward structure	PBRS with λ decay	Bipartite: sparse env + dense intrinsic (fixed α)
LLM query frequency	Low (plateau-triggered)	High (per-option completion)
Safety mechanism	Softmax budget, EMA, rollback	DAG in prompt, reflection, env freezing

6.9 QLoRA Fine-Tuning Pipeline

While Section 3.2 established the theoretical foundations of Quantized Low-Rank Adaptation, this section details the complete practical pipeline used to produce the specialized Commander adapter deployed throughout the HRL experiments evaluated in Chapter 7. As will be demonstrated, a surprisingly brief fine-tuning process on a modest synthetic dataset is sufficient to dramatically improve the Meta-Controller's decision quality.

6.9.1 Oracle Dataset Generation

The training dataset is generated synthetically to ensure exhaustive coverage of the crafting DAG. Exploratory successful runs from the agents helped build the dataset. A deterministic oracle systematically enumerates all meaningful inventory combinations and applies hardcoded logic to determine the optimal `(agent_0_option, agent_1_option)` assignment for each state. For example: "if no pickaxe and wood ≥ 1 and stone ≥ 1, then both agents should `CRAFT_PICKAXE`."

To prevent the model from over-indexing on trivially simple early-game states, the generated states are grouped into six progression phases (e.g., early collection, iron mining, endgame combat). Each phase is upsampled equally to ensure a balanced training signal.

Two augmentation techniques are applied to improve generalization:

1. **Stochastic Oracle Augmentation (SOA):** To prevent the model from overfitting to the perfectly logical inventory progressions generated by the script, a randomly selected inventory field is zeroed out with a 10% probability before querying the oracle. For

example, a valid mid-game state like `[iron: 1, pickaxe: 1]` might be corrupted to the physically impossible state `[iron: 1, pickaxe: 0]`. By passing this corrupted state to the oracle to generate the training label, the LLM is explicitly trained on how to gracefully recover from weird, out-of-distribution inventory combinations. This robustness is critical because observation delays and asynchronous race conditions during live RL rollouts frequently produce these mathematically inconsistent states.

2. **Reflection Augmentation (40% injection rate):** A synthetic hallucinated failure is prepended to the prompt: "*You previously chose [Incorrect Option]. It failed after [N] steps.*" The model must then output the correct option despite this misleading prior. While intended to teach the Commander to recover from logical errors, Section 7.3.3 demonstrates that this training signal inadvertently caused the model to assume *any* reported failure implies a logical mistake (the "Reflection Contradiction").

The final raw dataset contains **1,200 balanced examples** spanning all DAG phases and agent status combinations.

6.9.2 ChatML Formatting and Dataset Splits

The raw prompt and completion pairs are converted into Qwen's native ChatML format (Yang et al. 2025), structuring each example as a three-turn conversation (`system`, `user`, `assistant`). The system message contains the Commander's role description and DAG rules, the user message provides the current inventory state (and, for reflection examples, the failure context), and the assistant message contains the expected JSON output.

After a 90/10 train–validation split and a $2\times$ training set duplication (to ensure sufficient gradient updates given the small dataset), the final corpus comprises **2,160 training examples** and **120 validation examples**.

6.9.3 Training Configuration

Fine-tuning was performed on the local workstation's NVIDIA RTX 5070 GPU (12GB VRAM), as described in Section 4.5. The strict memory envelope forced several architectural compromises documented in Table 6.

Table 6: QLoRA fine-tuning hyperparameters. Values marked with † were constrained by the 12GB VRAM budget.

Category	Parameter	Value
Base Model	Architecture	Qwen 2.5 7B-Instruct
	Source	HuggingFace Hub (Wolf et al. 2020)
Quantization	Precision	4-bit NormalFloat (NF4)
	Compute dtype	bfloat16
	Double quantization	Enabled
LoRA Adapter	Rank r	16
	Scaling α	32
	Dropout	0.05
	Target modules	All 7 (q/k/v/o/gate/up/down)
Training	Epochs	1
	Batch size†	1
	Gradient accumulation	8 (eff. batch = 8)
	Learning rate	2×10^{-4}
	Scheduler	Cosine with warmup (10%)
	Optimizer	Paged AdamW 8-bit (Dettmers et al. 2022)
	Max sequence length†	1,024 tokens
	Gradient checkpointing†	Enabled
Infrastructure	Library	HuggingFace TRL (Werra et al. 2020)
	Adapter library	PEFT v0.19.1 (Mangrulkar et al. 2022)
	Logging	TensorBoard

A critical training detail is the application of *response-only loss masking*, implemented via the TRL library (Werra et al. 2020). By default, causal language models learn by predicting every subsequent word in a sequence, meaning they would normally compute an error gradient for the prompt itself. This wastes training capacity on memorizing instructions. To solve this, a custom data collator leverages PyTorch's native `ignore_index` mechanic: it sets the target labels of all system and user prompt tokens to -100. Mathematically, the Cross-Entropy loss function ignores any token with a -100 label, computing zero gradient for it. For example, in the sequence `[User] State is X [Assistant] {"opt": 1}`, every token preceding the JSON response is assigned a -100 label, while only the JSON characters receive valid target IDs. As a result, the model is penalized exclusively on its final JSON output, ensuring that every weight update directly improves its strategic decision-making rather than memorizing boilerplate text.

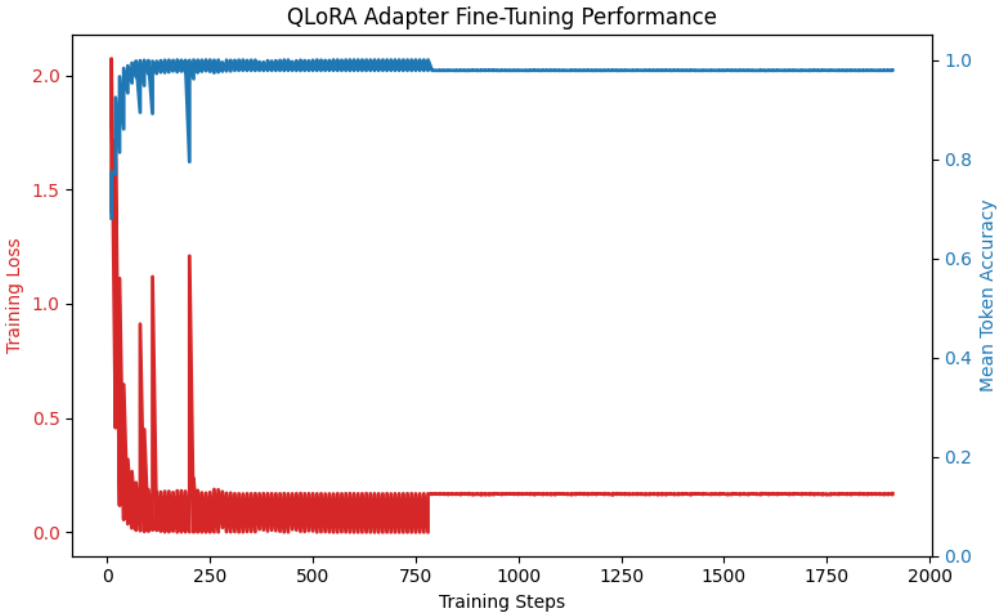
6.9.4 Training Results and Adapter Deployment

The complete fine-tuning process converged in approximately **3.1 hours** (1,910 steps) on the RTX 5070. As shown in Figure 18, the training loss dropped rapidly from 1.605 to 0.165, with the final evaluation metrics confirming near-perfect task acquisition:

- Evaluation loss: **0.0007**

- Mean token accuracy: **99.98%**

Figure 18: QLoRA adapter fine-tuning performance over 1,910 training steps. The training loss (left axis) converges rapidly within the first 500 steps, while the mean token accuracy (right axis) saturates near 100%.



The resulting LoRA adapter weighs only **154 MB**, approximately 1% of the full 15 GB base model, yet encodes the complete DAG reasoning and JSON formatting capabilities required by the Meta-Controller. For deployment during RL training, the adapter is loaded via the PEFT library and merged with the base model in memory, or alternatively converted to GGUF format and served through the Ollama inference framework (Section 4.5). Both pathways produce inference latencies compatible with the synchronous RL training loop, as quantified in Section 7.4.2. This fine-tuned adapter serves as the Meta-Controller for all “QLoRA” HRL configurations evaluated in Chapter 7.

6.10 Summary

The Meta-Controller approach establishes a strict separation of concerns: the LLM Commander dictates *what to do* (strategic option selection), while the MAPPO policy learns *how to do it* (kinetic execution). By appending a one-hot option encoding to the state vector, a single policy network learns option-specific behaviors. Simultaneously, the bipartite reward structure ensures the policy receives both dense intrinsic guidance and sparse environmental grounding.

Crucially, this architecture functions as an automated curriculum generator. Unlike traditional Curriculum Learning (Section 2.3), which requires manually designing environments of escalating difficulty, the Meta-Controller orchestrates a skill curriculum dynamically. It guides agents from basic resource gathering to advanced combat coordination entirely within a single, static environment.

Table 7: Key hyperparameters of the Meta-Controller approach.

Component	**Parameter**	**Value**
Options	$\lvert\mathcal{O}\rvert$	10
Staleness	T_{stale}	150 steps
Cooldown	Post-query cooldown	50 steps
Intrinsic reward	Distance scale λ_{intr}	0.05
	Success bonus	$+1.0$
Composite reward	α_{intr}	0.5
	$D_{\max}$	85.0
Target KL	KL threshold	0.015
Network	One-hot dimension	10
	Total `vec_dim`	25

7 Experiments and Results

This chapter evaluates the hypothesis that a Meta-Controller LLM can resolve exploration bottlenecks in a sparse Multi-Agent Reinforcement Learning (MARL) environment through dynamic reward shaping, and subsequently, how Hierarchical Reinforcement Learning (HRL) further improves these capabilities.

7.1 Experimental Setup

To evaluate the proposed architectures, we compare distinct configurations within our multi-agent resource-gathering environment, structured into two parts.

For the **Dynamic Reward Shaping** experiments, we evaluate:

1. **Sparse Baseline:** A standard MAPPO implementation (Yu et al. 2022) relying on sparse environmental rewards (+1 for completing milestones).
2. **LLM Dynamic (Gemini 2.5 Flash Lite):** The proposed "Mixing Board" architecture, utilizing Google's Gemini 2.5 Flash Lite model via API as the Commander to dynamically shift potential-based reward weights.
3. **LLM Dynamic (Local Qwen 2.5 7B):** An equivalent dynamic architecture utilizing a localized, smaller parameter model (Qwen 2.5 7B) to observe the efficacy of smaller, self-hosted LLMs.

For the **Hierarchical Reinforcement Learning (HRL)** experiments, we evaluate four configurations:

1. **No-LoRA Base with Reflection (Qwen 2.5 7B):** The un-finetuned base model serving as the HRL Meta-Controller, with counterfactual reflection triggered on option staleness.
2. **No-LoRA Base without Reflection (Qwen 2.5 7B):** The same un-finetuned base model, with the reflection mechanism disabled.
3. **QLoRA with Reflection (Qwen 2.5 7B):** A QLoRA fine-tuned model (as detailed in Section 6.9, the Qwen 2.5 7B model was fine-tuned using a 4-bit NF4 quantization pipeline on 1,200 synthetic oracle examples) with counterfactual reflection triggered on option staleness.
4. **QLoRA without Reflection (Qwen 2.5 7B):** The same fine-tuned model, with the reflection mechanism disabled.

All runs were executed with identical MAPPO hyperparameters (clipping $\epsilon = 0.2$, GAE[4] $\lambda = 0.95$, $\gamma = 0.99$, across 128 parallel environments).

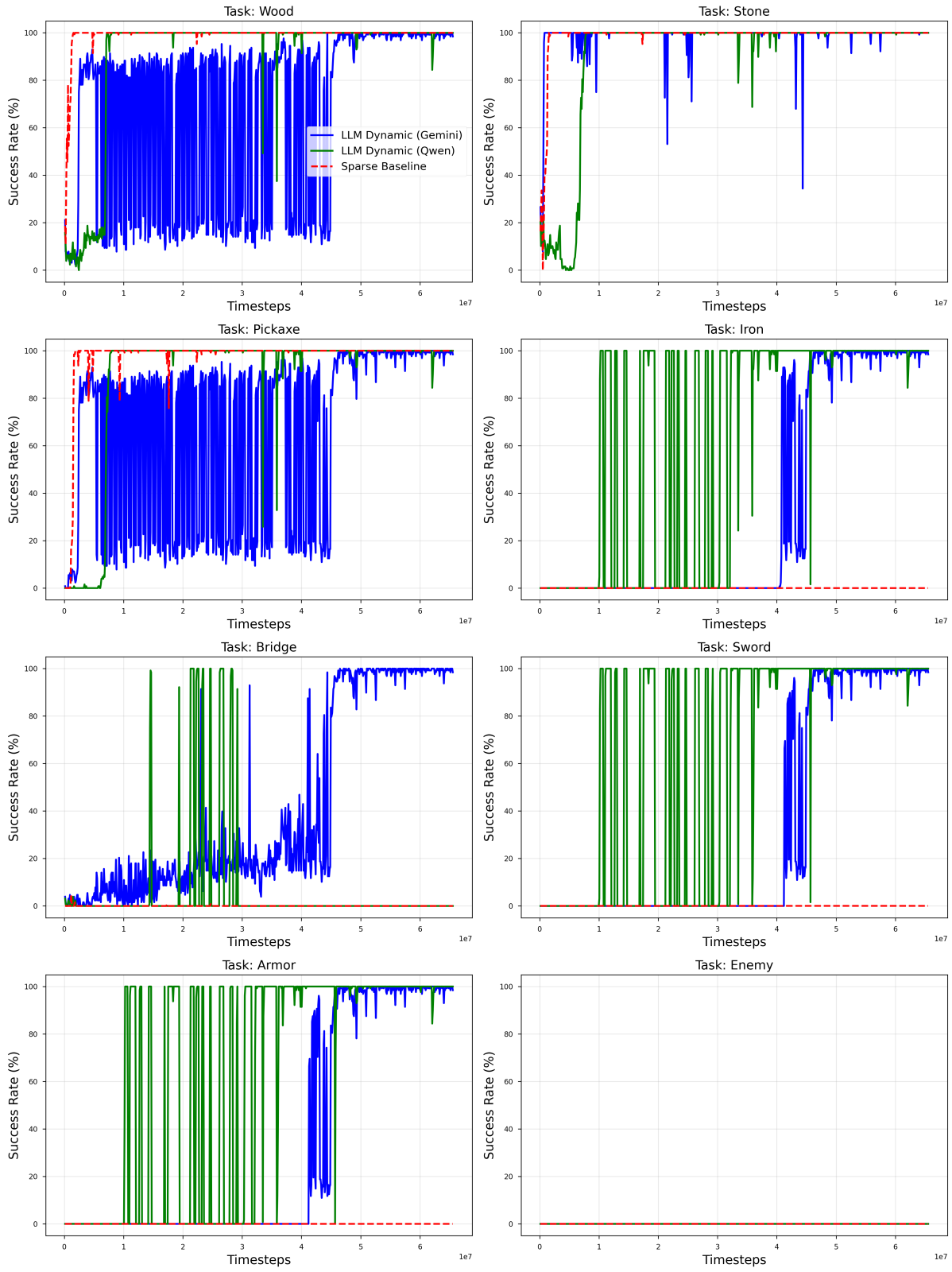
Finally, to evaluate the structural stability and adaptability of the system, we conduct targeted ablation studies (such as the necessity of EMA smoothing) and a **Semantic Shift** experiment, detailing the Meta-Controller's zero-shot ability to adapt to sudden environmental rule changes without retraining. These are explored in Part 3 (Section 7.4).

[4] Generalized Advantage Estimation

7.2 Part 1: Dynamic Reward Shaping

The progression through the environment's strict dependency Directed Acyclic Graph (DAG) serves as the primary metric to validate the LLM Dynamic architecture.

Figure 19: Subtask Completion Rates: Comparison between the Sparse Baseline and the LLM Dynamic models over 2000 training updates.



As shown in Figure 19, all three configurations solve the early-game tasks (Wood, Stone) initially. The Pickaxe subtask follows shortly after, with the Baseline reaching it at step 1.1M and Gemini at step 720K.

However, the Baseline fails at the *Iron* bottleneck. Without dynamic reward shaping to guide exploration, the Baseline workers do not discover how to consistently mine Iron (0% final rate), preventing the learning of downstream tasks (Sword, Armor, Bridge).

In contrast, both LLM-guided runs overcome the Iron bottleneck at different speeds. **Qwen** unlocks Iron at step 10M, and discovers Sword and Armor around step 10.1M. This rapid progression is explained by the DAG structure: once Iron is available, Sword and Armor only require visiting the Workbench, a behavior the agents have already mastered. **Gemini 2.5 Flash Lite** unlocks Iron later, at step 40.7M. Once unlocked, an identical cascade occurs, with Sword and Armor appearing at step 41.3M.

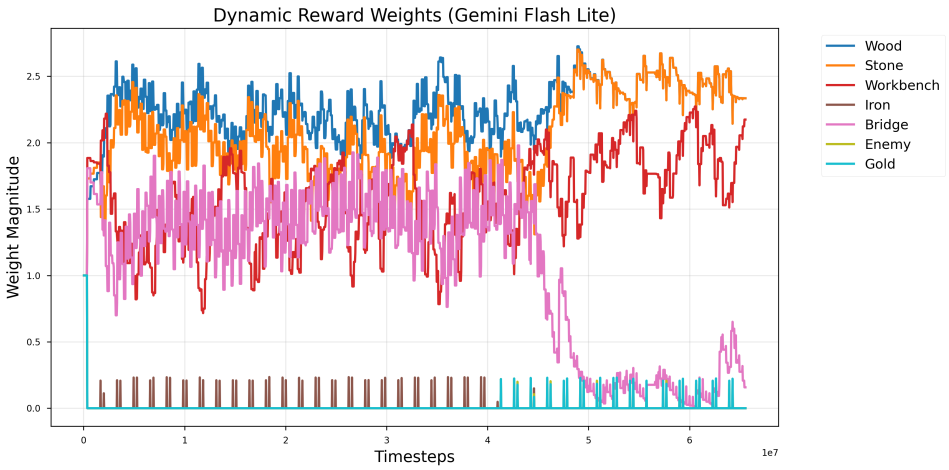
The learning dynamics for both LLM configurations are characterized by high variance. Rather than smooth monotonic improvement, the agents exhibit repeated oscillations in task completion rates across many updates before stabilizing on a subtask. This is particularly visible in the mid-game tasks (Pickaxe, Bridge), where completion rates fluctuate between 0% and partial success for extended periods. In the case of **Qwen**, the agents eventually forget how to build a Bridge after their initial success around step 2.97×10^7, regressing to 0% completion. This indicates that the learned policies can remain fragile when the shaping signal shifts focus away from previously mastered behaviors.

Neither run achieved Enemy defeats (0% for all configurations), indicating that the combat subtask represents a challenge beyond reward shaping. It likely requires specific action coordination that the current scalar reward signal cannot provide.

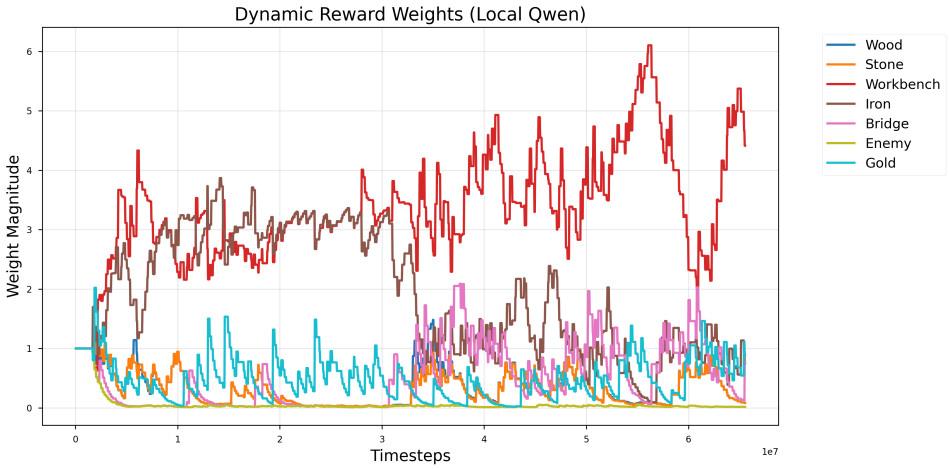
7.2.1 LLM Interventions in Action

To understand how the LLMs shaped exploration, we analyse the evolution of the reward weights. The weights shown in Figure 20 are the *post-softmax* weights actually applied to the potential-based shaping (Equation 20).

Figure 20: Evolution of the Dynamic Reward Weights (post-softmax, as applied to the PBRS signal).



(a) Gemini 2.5 Flash Lite Meta-Controller (198 interventions)



(b) Local Qwen 2.5 7B Meta-Controller (20 interventions)

Figure 20 shows two distinct command strategies, though both share an initial semantic understanding: from the very first intervention, both Commanders set $w_{\text{enemy}} = 0.0$ and $w_{\text{gold}} = 0.0$ throughout the entire run. Both models recognise that these terminal tasks are unreachable early on and allocate no reward budget to them.

Beyond this shared starting point, the strategies diverge. **Gemini's strategy** employs oscillation. It alternates between maximizing early tasks (w_{wood}, w_{stone}) and late tasks (w_{bridge}). However, because the agents had not yet mastered Pickaxe, the DAG guardrail layer (Section 5.2) programmatically masked out the late-game weights to 0.0. By constantly shifting its feasible weights, Gemini caused the greedy planner's target to rapidly bounce between Wood and Workbench. This broke the agents' focus and slowed their progression toward Iron.

Qwen's strategy is highly stable. Rather than micromanaging the agents' immediate next steps, Qwen effectively establishes a consistent, global hierarchy of strategic intent ($w_{\text{wood}} \approx 2.33$, $w_{\text{stone}} \approx 2.33$, $w_{\text{workbench}} \approx 1.51$, $w_{\text{iron}} \approx 0.38$) and delegates the tactical execution to the DAG planner.

This stability perfectly leverages how the greedy planner selects exactly one target subtask ($g^{(i)} = \arg\max_{j \in \mathcal{F}} R_j^{\text{base}} \cdot w_j$). Initially, Wood and Stone yield the highest scores ($2.33 \times 2.0 = 4.66$). Because Qwen's weights provide a static, ordered priority list, subtask switching is driven entirely by the DAG mask: once Wood and Stone are mastered, they are removed from the feasible set $\mathcal{F}$, automatically shifting the planner's target to the Workbench ($1.51 \times 3.0 = 4.53$). Once the Pickaxe is crafted, Iron enters $\mathcal{F}$ and seamlessly becomes the new target ($0.38 \times 2.0 = 0.76$).

This explains the intervention frequency disparity (198 for Gemini vs. 20 for Qwen). The *CriticTrigger* (Section 5.3) fires only during performance plateaus. Qwen's static hierarchy allowed the DAG to naturally guide the planner, resolving the Iron bottleneck by step 10M and keeping the trigger closed. Gemini's oscillation kept the target unstable, stranding agents at the Iron bottleneck until step 40.7M and causing constant trigger fires. Gemini's high intervention count is thus a symptom of stagnation, not proactive command.

Ironically, Qwen succeeded here by acting exactly like a pre-coded topological sort, letting the DAG rules pull the agents forward. In a predictable environment, dynamic LLM interventions are largely unnecessary and even detrimental (as seen with Gemini). The true necessity of an LLM planner is evaluated later in the environmental shift experiments. Its core asset is the ability to adapt when predefined rules break down or semantics shift.

7.2.2 Reward Analysis

Figure 21: Average Environment Reward (True Milestones Reached).

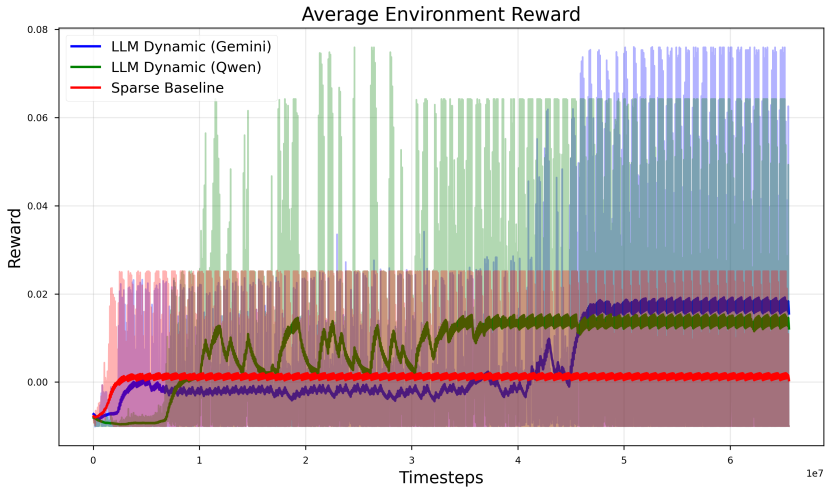


Figure 21 presents the true average environment reward. Both LLM Dynamic runs achieve higher returns than the Baseline, reaching an identical peak reward (≈0.076). This indicates that once the bottleneck is resolved, the final quality of the learned policy is comparable; the primary difference lies in the *sample efficiency* driven by Qwen's stable weight profile.

However, the training trajectories highlight a trade-off in policy stability (Figure 19). Around step 2.97×10^7, the Qwen-guided agents temporarily forget how to build a Bridge, causing a dip in average reward. Gemini's policy is much more stable; once the agents learn Bridge, they retain it, producing a higher reward spike at step 4.65×10^7. This stability is a direct benefit of Gemini's broad amplification: by constantly reinforcing prerequisite tasks (wood, stone), Gemini provides

a safety net against forgetting. Qwen's focused suppression of mastered tasks removes this safety net, leaving the policy vulnerable when the reward signal shifts to new goals.

7.3 Part 2: Hierarchical Reinforcement Learning

While the dynamic shaping architecture successfully guided the lumberjack and miner past initial bottlenecks, it ultimately stalled at the combat phase. Modifying the continuous reward landscape was insufficient to teach the workers the discrete, long-horizon macro-actions required for coordinated combat.

This limitation motivated the shift to a full Hierarchical Reinforcement Learning (HRL) architecture, where the Commander explicitly assigns discrete *Options* directly to the lower-level MAPPO policy. To systematically investigate the exploration bottleneck and the impact of our proposed mechanisms, we evaluated four configurations over 2000 optimization updates ($\approx$65.5M environment steps each): the un-finetuned **No-LoRA Base** (with and without counterfactual reflection) and the **QLoRA** fine-tuned variant (with and without counterfactual reflection). The reflection mechanism is triggered whenever an assigned Option becomes "stale" (i.e., workers fail to complete it within 150 steps).

7.3.1 Subtask Progression

All four configurations managed to complete the DAG at some point, but with significant differences in efficiency and stability. Table 8 presents the exact environment step at which each subtask was first completed.

Table 8: First breakthrough step (millions of environment steps) for each subtask. **Bold** indicates the fastest configuration per subtask.

Subtask	No-LoRA + Refl.	No-LoRA – No Refl.	QLoRA + Refl.	QLoRA – No Refl.
Wood	0.16M	0.26M	0.29M	0.26M
Stone	0.16M	0.26M	0.29M	0.26M
Pickaxe	13.50M	5.01M	1.25M	**0.72M**
Bridge	30.61M	5.73M	2.03M	**1.18M**
Iron	16.61M	10.52M	**2.13M**	3.01M
Sword	16.71M	10.52M	**2.29M**	3.15M
Armor	16.78M	10.52M	**2.29M**	3.15M
Enemy	30.74M	11.89M	**2.49M**	3.24M
Gold	32.31M	16.32M	7.08M	**6.29M**

Several critical patterns emerge from Table 8. First, the QLoRA models show a massive overall advantage: both fine-tuned runs complete the mid-game milestones (Pickaxe through Armor) approximately 8–14$\times$ faster than the baseline equipped with reflection. At the final Gold milestone, QLoRA without Reflection is the fastest (6.29M), closely followed by QLoRA with Reflection (7.08M).

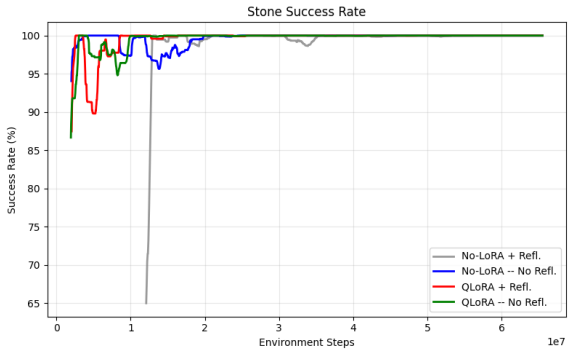
However, the most revealing finding relates to the reactive self-debugging failure mode. When counterfactual reflection is applied to the base No-LoRA model, it acts as a severe destabilizing force rather than a helpful correction. The No-LoRA model without reflection reaches the Gold

milestone in 16.32M steps, while the No-LoRA model with reflection takes 32.31M steps, nearly twice as long.

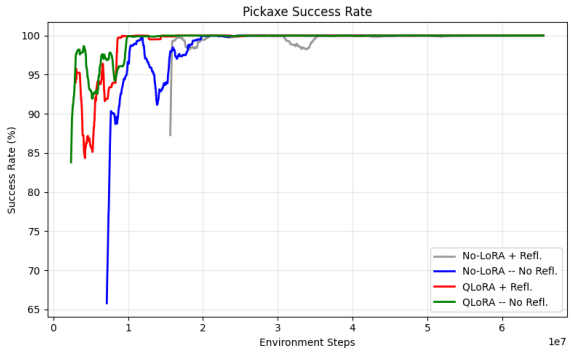
This occurs because the un-finetuned model lacks the specific domain reasoning to interpret the reflection prompt, “Agent 0 was assigned "[Option]" and Agent 1 was assigned "[Option]". After N steps, NEITHER agent has succeeded.” Instead of correcting its logic, the base model reacts erratically and rapidly switches its assigned options, constantly changing the intrinsic reward signal for the RL agents and heavily destabilizing the PPO training. Conversely, without reflection, the base model falls into semantic over-confidence, repeatedly re-confirming the same option assignment regardless of context. Ironically, this rigidity is better for the MAPPO agents: it creates a more stationary environment, allowing the low-level policy to learn a stable behavior conditioned on a fixed (even if suboptimal) option signal. This provides strong empirical evidence that reactive self-debugging is a double-edged sword: it is effective when paired with a fine-tuned model, but actively harms convergence when applied to a base model.

Figure 22: Early subtask completion rates for the four HRL configurations.

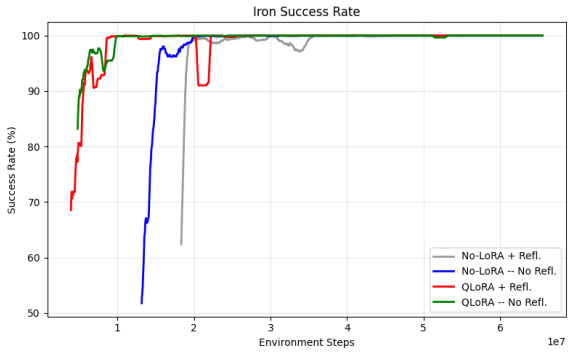
(a) Wood



(b) Stone

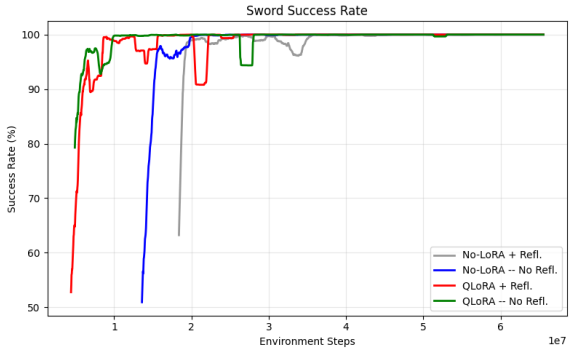


(c) Pickaxe

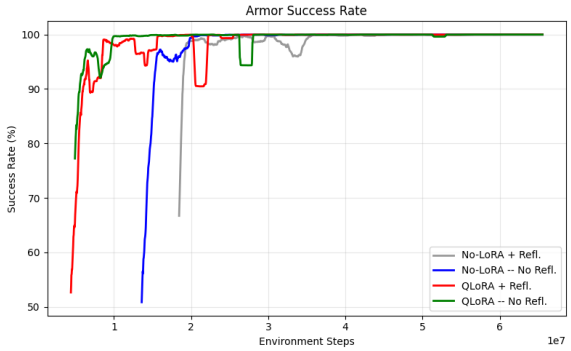


(d) Iron

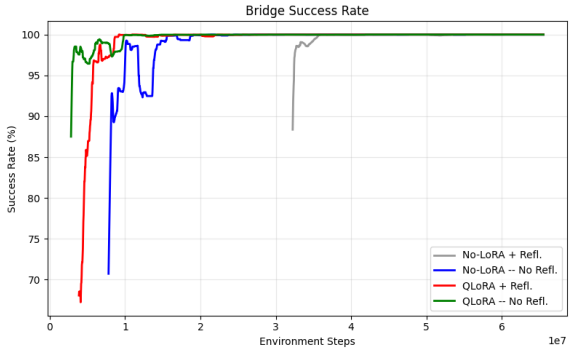
Figure 23: Late subtask completion rates across the remaining environment milestones for the four HRL configurations.



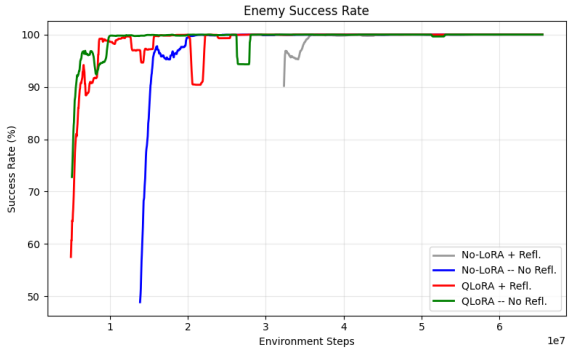
(a) Sword



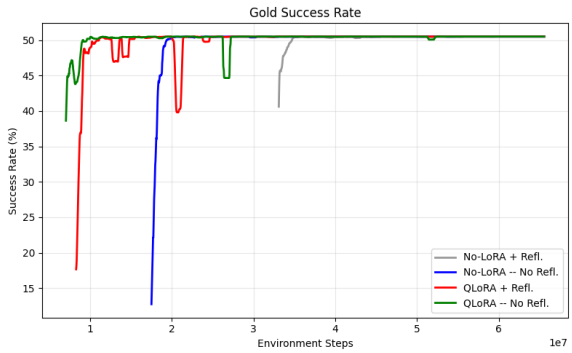
(b) Armor



(c) Bridge



(d) Enemy



(e) Gold

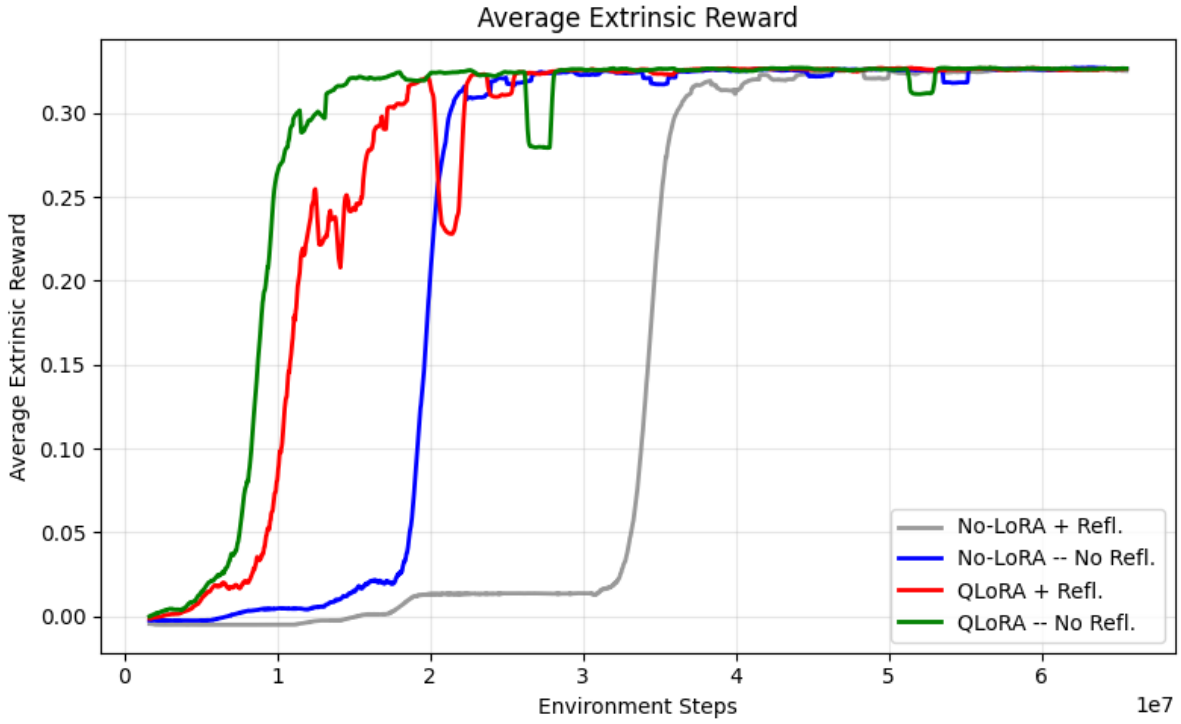
Figures 22 and 23 show that all four configurations solve the complete DAG, reaching 100% completion on Gold. This is a significant improvement over the Dynamic Shaping architecture (Part 1). The key differentiator is sample efficiency: QLoRA without Reflection reaches 100% on Gold by $\sim$8M steps, QLoRA with Reflection by $\sim$10M steps, the No-LoRA base without reflection by $\sim$16M steps, and the No-LoRA base with reflection requires $\sim$35M steps (consuming over half the training budget).

The subtask curves also reveal important differences in learning stability. The most severe volatility is exhibited by the No-LoRA configurations (both with and without reflection), which show deep, recurring regressions even after a subtask is ostensibly mastered. For instance, in the Sword, Iron, and Enemy plots (Figures 22 and 23), the No-LoRA runs plummet from

near-100% success rates before slowly recovering. This pattern is correlated with their delayed learning onset: because the un-finetuned Commander provides poor initial option sequencing, the PPO policy begins learning later and in a more volatile intrinsic reward landscape. By contrast, both QLoRA configurations exhibit faster initial ramps and fewer deep regressions, confirming that fine-tuning the Commander is the dominant factor in learning stability. Within the QLoRA pair, the No-Reflection variant shows slightly smoother curves, consistent with the hypothesis that a deliberate, unwavering option sequence provides a more stationary intrinsic reward landscape for the low-level workers to optimize against.

7.3.2 Reward Trajectories

Figure 24: Average Extrinsic Reward across HRL configurations. All four configurations converge to comparable final rewards ($\sim$0.326). Fine-tuning accelerates learning (QLoRA converges faster than No-LoRA), while the reflection mechanism consistently delays convergence across both.



The reward trajectories (Figure 24) quantify the sample efficiency gap. All four configurations converge to nearly identical final rewards $\sim$0.326, with peak rewards clustered at $\sim$0.335. This convergence confirms that the underlying PPO policy has sufficient capacity to master the environment regardless of the Meta-Controller quality, given enough time. The practical difference is entirely one of sample efficiency.

Compared to the Dynamic Shaping architectures (Part 1), which plateaued at 0.076, the HRL architecture achieves a **4.4$\times$ improvement** in peak reward. When analyzing sample efficiency to first reach the peak reward plateau ($\sim$0.326), a consistent pattern emerges: fine-tuning accelerates learning, while reflection delays it. The QLoRA without Reflection run is the fastest, reaching the plateau by $\sim$13M steps. Adding reflection to QLoRA delays this to $\sim$19M steps. A similar gap exists for the un-finetuned models: the No-LoRA without Reflection run reaches the

plateau by ∼22M steps, whereas the No-LoRA with Reflection run is severely delayed, flatlining near zero reward until finally reaching the plateau around ∼37M steps. This consistent sample efficiency gap is a clear quantitative signal that the reflection mechanism actively delays learning in terms of environment steps. Furthermore, because generating counterfactual reflections requires outputting additional tokens, the "With Reflection" runs also incur a higher wall-clock inference overhead compared to their non-reflective counterparts.

7.3.3 KL-Divergence Guardrail

In the HRL architecture, the transition between discrete macro-actions (Options) induces abrupt shifts in the intrinsic reward landscape. From the perspective of the low-level PPO workers, the objective function is highly non-stationary. To prevent catastrophic forgetting of previously learned kinetic policies, we bound the PPO surrogate objective update using a strict Kullback-Leibler (KL) divergence guard.

During the optimization epochs, we continuously monitor the approximate KL divergence between the behavioral policy $\pi_{\theta_{\text{old}}}$ and the updating policy π_θ (Schulman et al. 2017):

$$D_{\text{KL}}^{\text{approx}} = \mathbb{E}\left[-\log\left(\frac{\pi_\theta(a|s,o)}{\pi_{\theta_{\text{old}}}(a|s,o)}\right)\right] \tag{35}$$

If $D_{\text{KL}}^{\text{approx}}$ exceeds the predefined trust region threshold ($\delta = 0.015$, implemented as a target KL divergence), the gradient updates for the current batch are immediately aborted.

Figure 25: Maximum approximate KL divergence per update across the four HRL configurations. The dashed red line marks the early-stopping threshold $\delta = 0.015$.

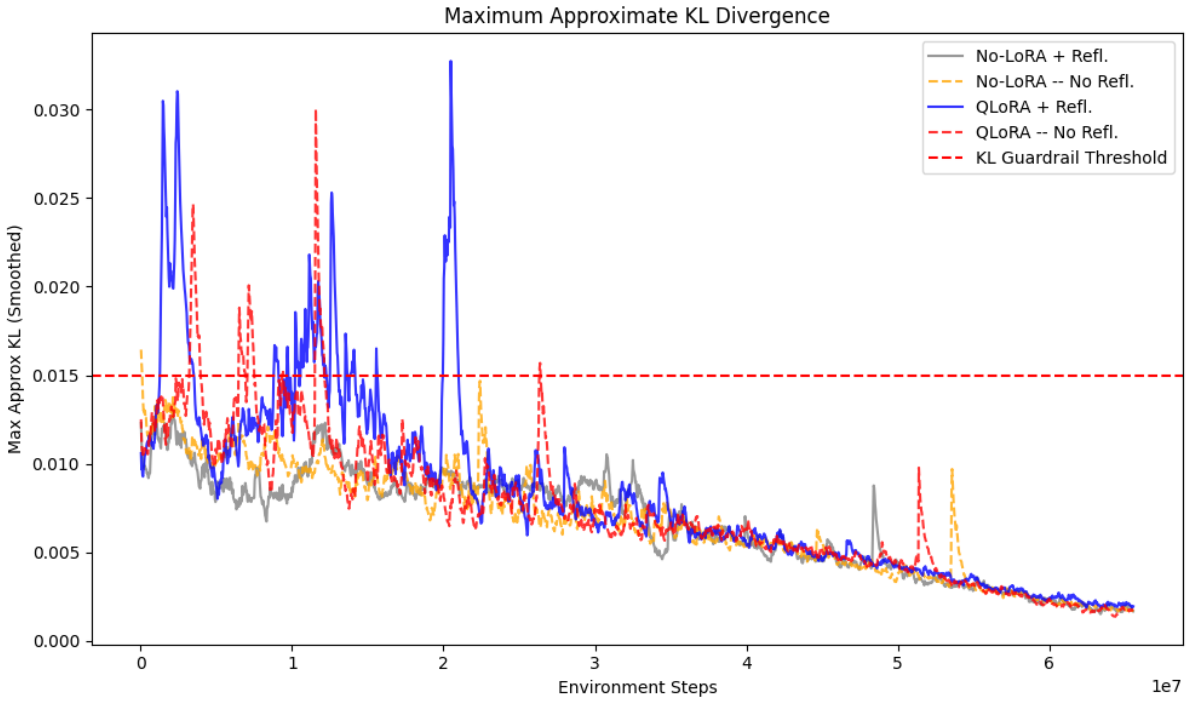


Table 9: KL divergence statistics and policy stability metrics across HRL configurations.

Metric	No-LoRA + Refl.	No-LoRA – No Refl.	QLoRA + Refl.	QLoRA – No Refl.
Mean $\max D_{\text{KL}}$	0.0066	0.0067	0.0082	0.0073
Peak $\max D_{\text{KL}}$	0.025	0.042	**0.117**	0.107
KL early-stop triggers	46 / 2,000	55 / 2,000	**234** / **2,000**	152 / 2,000
Staleness resets (total)	28,927	78,535	**88,711**	39,137
LLM queries dispatched	12,752	**14,036**	12,791	13,710
Final TD error (last 100 updates)	0.042	0.032	0.024	**0.023**
Mean TD error (all updates)	0.046	0.049	0.060	0.048

Table 9 highlights how KL instability disrupts learning. The QLoRA with Reflection run hits the KL guardrail **5.1**$\times$ **more often** than the No-LoRA + Refl. base and **1.5**$\times$ **more often** than the QLoRA – No Reflection run. Since each KL trigger immediately aborts the current gradient update, this constant disruption prevents the PPO workers from smoothly optimizing their kinetic behaviors. This manifests directly as the jagged fluctuations seen in its reward trajectory (Figure 24).

Furthermore, the data reveals a fascinating divergence in how reflection impacts sample efficiency depending on the underlying model's capability. For the fine-tuned QLoRA model, reflection actively degrades efficiency: the QLoRA + Reflection run accumulates **2.3**$\times$ **more staleness resets** than its No-Reflection counterpart (88,711 vs. 39,137). When the fine-tuned model is told a plan failed, the reflective Commander tends to overcorrect or generate a worse plan that fails even faster, causing rapid cycling of short-lived Options. Conversely, for the weaker No-LoRA model, reflection *improves* stability: the No-LoRA – No Reflection run suffers **2.7**$\times$ **more staleness resets** (78,535 vs. 28,927) than when reflection is enabled. This suggests that while reflection can help a weaker baseline model avoid getting permanently stuck, it interferes with the already-optimized priors of a fine-tuned model.

This paradox originates from a misalignment between the instruction-tuning data and actual RL inference. During QLoRA fine-tuning, the dataset injected synthetic counterfactuals mimicking the environment's reflection prompt (e.g., *"Agent 0 was assigned [Option] and Agent 1 was assigned [Option]. After [N] steps, NEITHER agent has succeeded."*) to teach the model to correct logical errors. Consequently, the fine-tuned Commander appears to have developed an unintended bias: it overwhelmingly interprets any failure prompt as evidence that its previous choice was logically flawed and must be changed.

During RL training, however, option "staleness" rarely indicates a logical error. More often, it represents a *kinetic delay*: the agents simply need more time to navigate the terrain or collect resources for a fundamentally correct plan. When the reflection prompt fires due to this kinetic delay, the Commander incorrectly assumes a reasoning error and abandons the correct long-horizon option.

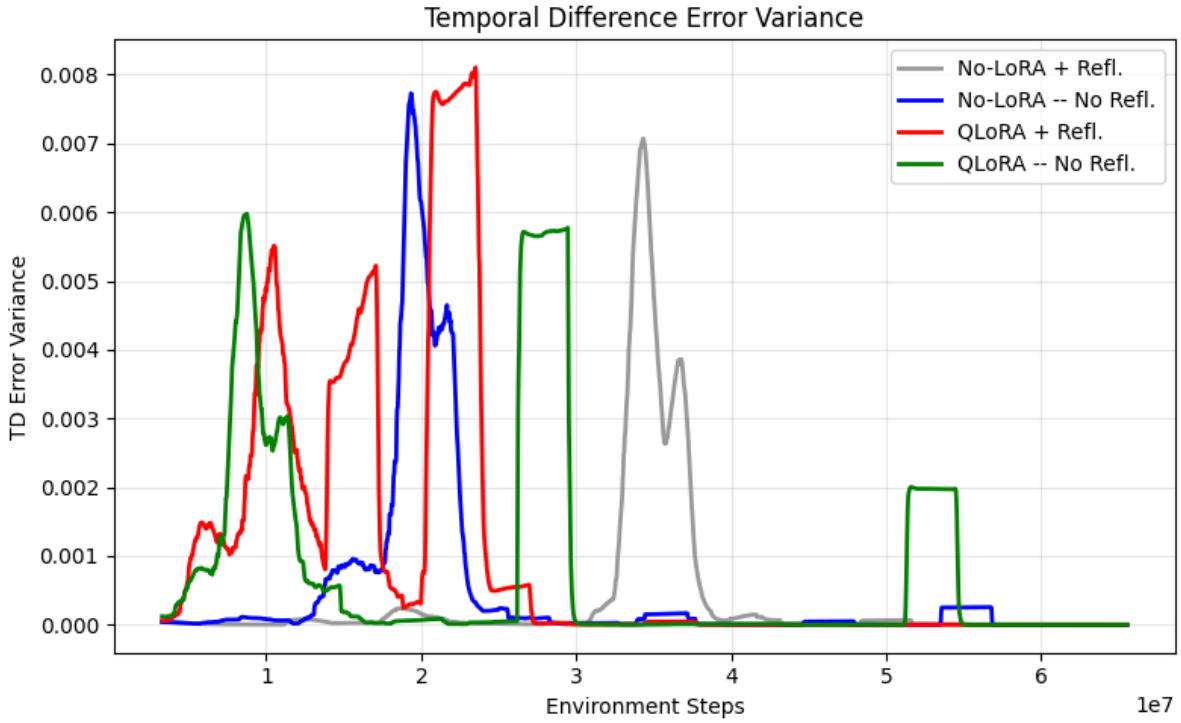
To illustrate this, consider a snapshot from the evaluation logs. The workers have defeated the Enemy and need to collect Gold. The Commander correctly assigns `COLLECT_GOLD`, but when the workers take too long, staleness triggers the reflection prompt:

```
Inventory: {..., 'pickaxe': 1, 'sword': 1, 'bridge': 1, 'enemy': 1, 'gold': 0}
Previous Assignment: Agent 1 failed on COLLECT_GOLD
```

```
LLM Response:
{
  "reflection": "Previous assignment of IDLE and COLLECT_GOLD failed because
                 agents lacked the necessary sub-tier items.",
  "dag_check": "A0 waits (bridge done), A1 crafts armor.",
  "agent_0_option": "IDLE",
  "agent_1_option": "CRAFT_ARMOR"
}
```

Because the prompt implies a mistake, the Commander hallucinates a logical error ("lacked necessary items") and regresses to an irrelevant prerequisite (`CRAFT_ARMOR`). Disabling reflection prevents this forced doubt, allowing the Commander to stably re-confirm the correct Option.

Figure 26: Temporal Difference Error Variance throughout the training for all HRL configurations.



The downstream effect is quantified by both the TD Error variance (Figure 26) and the mean TD error statistics (Table 9). Figure 26 shows that each configuration exhibits a distinct variance spike during its active learning phase: the QLoRA runs spike earlier (5–25M steps) and settle quickly, while the No-LoRA + Refl. run produces a delayed but large spike around 30–40M steps, coinciding with its late-onset learning. The mean TD error statistics from Table 9 confirm this pattern: the QLoRA + Reflection run maintains the highest mean TD error across the full training run (0.060), 25% higher than the QLoRA No-Reflection run (0.048), reflecting the Critic's inability to track the rapidly switching intrinsic reward landscape caused by reflection-induced option churn. However, as the policies saturate and option switching diminishes, the TD errors stabilize. Averaged over the final 100 updates, both QLoRA runs converge to a comparable mean TD error ($\sim$0.024). The No-LoRA configurations settle at higher final mean TD errors (0.042 and

0.032), as the Critic continues to struggle with the less structured option transitions generated by the un-finetuned model.

Ultimately, all four HRL configurations solve the complete DAG. The key findings are twofold: first, QLoRA fine-tuning significantly accelerates sample efficiency ($5.1\times$ faster to Gold); second, while the KL-divergence guardrail absorbs major non-stationarity, it cannot filter out the high-frequency, sub-threshold perturbations caused by the reflection loop. Thus, forcing reflection upon a fine-tuned model actively delays convergence.

7.4 Part 3: Robustness and Semantic Adaptation

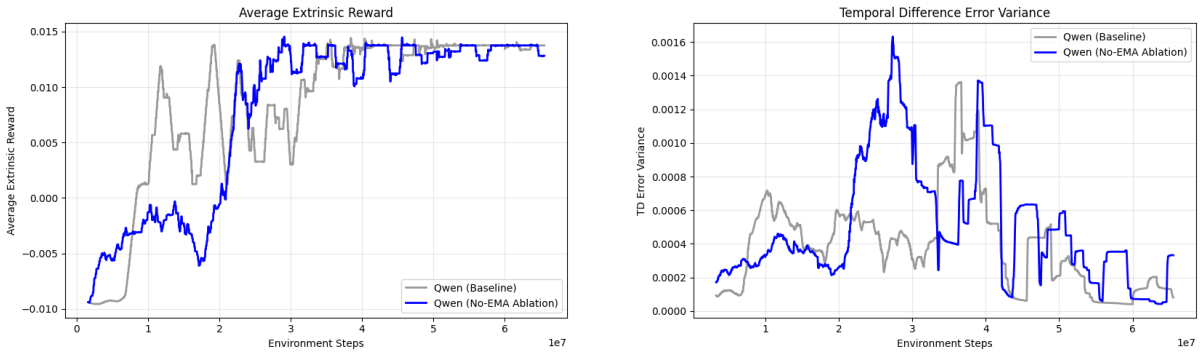
HRL outperforms Dynamic Shaping for solving the complete DAG. Before concluding, we return to the Mixing Board architecture to validate two remaining claims: the necessity of temporal smoothing, and the LLM's zero-shot semantic adaptation.

7.4.1 Ablation: The Necessity of EMA Smoothing

In Section 5.5.1, we posited that feeding raw, discrete priority logits directly into the potential-based shaping function would induce severe non-stationarity. To empirically verify this, an ablation run was conducted wherein the Exponential Moving Average (EMA) layer was disabled.

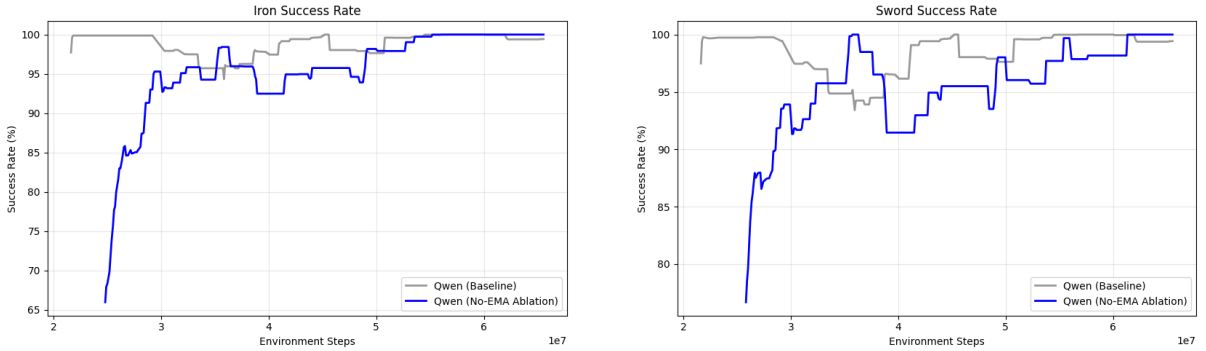
As shown in Figure 27, removing the EMA filter results in a noticeable spike in TD-error variance. The ablation itself was executed by bypassing the exponential smoothing step entirely in the programmatic orchestration layer.

Figure 27: Comparison of training stability with and without EMA smoothing ($\alpha = 0.2$). Disabling smoothing induces a severe spike in Temporal Difference (TD) error variance (right).



The Critic network struggles to track a reward objective that shifts instantaneously. Figure 28 illustrates the behavioral consequence: the non-stationarity leads to suboptimal optimization, causing the workers to exhibit slight regressions in their `Iron` and `Sword` completion rates during the mid-stages of training. However, it is worth noting that the No-EMA workers do eventually recover and climb back to a 100% success rate on these tasks. Thus, while the workers can eventually overcome the noise, this confirms that the EMA smoothing equation is highly helpful for bridging discrete LLM outputs with continuous RL gradients, enabling faster convergence and preventing such regressions.

Figure 28: Subtask completion rates for Iron (left) and Sword (right). The variance spike directly correlates with suboptimal learning and temporary regressions in advanced subtasks.

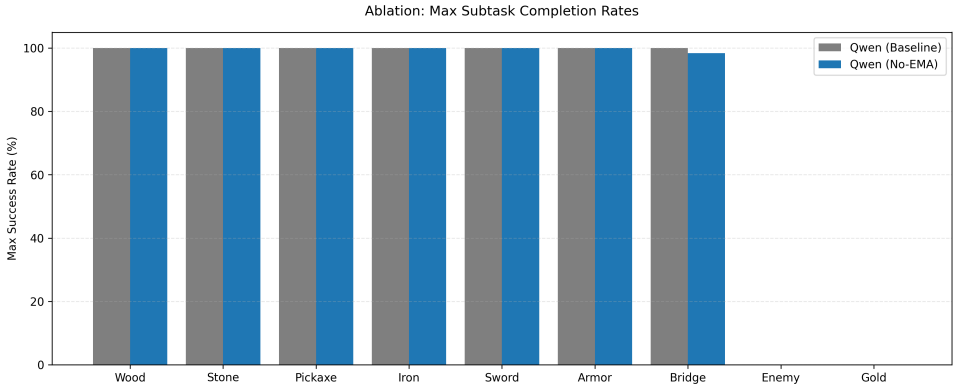


To quantify the optimization efficiency provided by the EMA filter, Table 10 details the number of environment steps required to reliably learn (i.e., achieve a $\geq 90\%$ success rate) the advanced subtasks.

Table 10: Environment steps required to reach a 90% completion rate.

Subtask	**With-EMA (Baseline)**	**No-EMA (Ablation)**
`Wood`	0.33M steps	2.36M steps ($7.2\times$ slower)
`Stone`	0.82M steps	2.46M steps ($3.0\times$ slower)
`Pickaxe`	1.31M steps	6.88M steps ($5.3\times$ slower)
`Bridge`	2.06M steps	9.24M steps ($4.5\times$ slower)
`Iron`	2.22M steps	21.00M steps ($9.4\times$ slower)
`Sword`	2.52M steps	21.63M steps ($8.6\times$ slower)
`Armor`	2.52M steps	21.63M steps ($8.6\times$ slower)
`Enemy`	Failed to Converge	Failed to Converge
`Gold`	Failed to Converge	Failed to Converge

Figure 29: Maximum subtask completion rates achieved across all runs. Despite the early variance spikes, No-EMA eventually masters all tasks identical to the baseline.



7.4.2 Computational, Financial, and Energy Overheads

Integrating LLMs into the RL training loop introduces severe wall-clock overhead if queries rely on external cloud APIs. This latency makes commercial cloud models impractical for high-frequency control.

To quantify this, a standard 128-update training run using a traditional programmed meta-controller (the Standard HRL Baseline) finished in **8.30 hours**. Routing those same decisions through a cloud API (Gemini 1.5 Flash) increased the training time to **12.27 hours**. This 48% slowdown is caused entirely by network latency. Testing heavier proprietary models like GPT-4 was infeasible due to API rate limits.

Our optimized implementation uses a smaller, locally fine-tuned Qwen2.5-7B model to solve this bottleneck. By eliminating network communication, the local model completed the run in **7.78 hours**, matching the speed of the programmed baseline.

This local optimization provides a double win in terms of resource efficiency. First, it eliminates the financial cost and power consumption associated with querying millions of tokens on commercial cloud servers. Second, it reduces the local workstation's own electricity footprint by nearly 40% simply by shortening the total training time from 12.27 hours to 7.78 hours.

7.4.3 Environment Shift: The Case for a Semantic Commander

A valid question is why we should use an LLM Commander at all. The previous results showed that a stable hierarchy paired with a DAG mask operates perfectly in a static environment. If the goal is simply to shift weights along a predictable path, a hard-coded Python script would be much more efficient.

The limitation of a programmed script is its inflexibility. It relies entirely on fixed rules and exact string matching. If a map designer renames the `Workbench` to `Furnace`, a hard-coded script will crash instantly with a missing key error. This highlights the core asset of the LLM. It possesses zero-shot semantic reasoning. The LLM understands the underlying context of the environment and can adapt to linguistic or structural changes dynamically, requiring absolutely no code modifications.

To test this adaptability, we injected an alias mapping into the orchestrator at Update 500. The `Workbench` was renamed to `Furnace` in the Commander's prompt space. A secondary control run replaced it with a nonsense string: `Zylorg`.

Figure 30: Subtask completion rates during semantic shifts. The `Furnace` shift traps workers in a local optimum, halting `Iron` and `Sword` progression, while the `Zylorg` shift organically recovers.

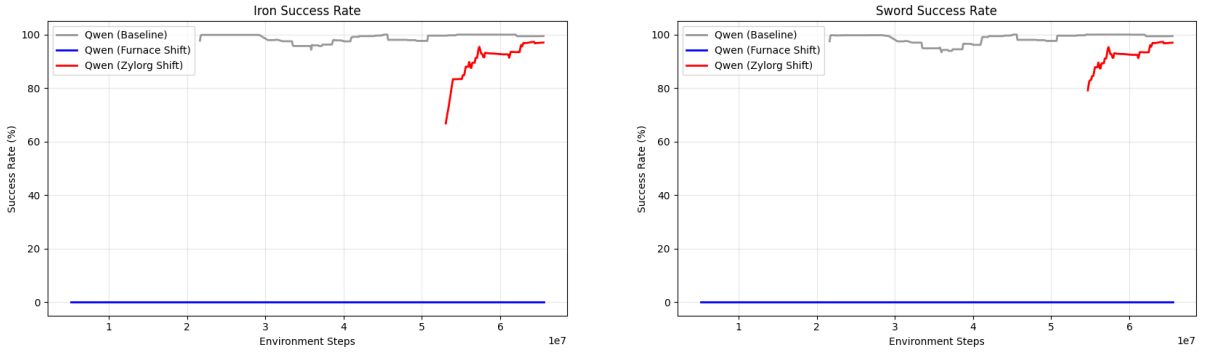
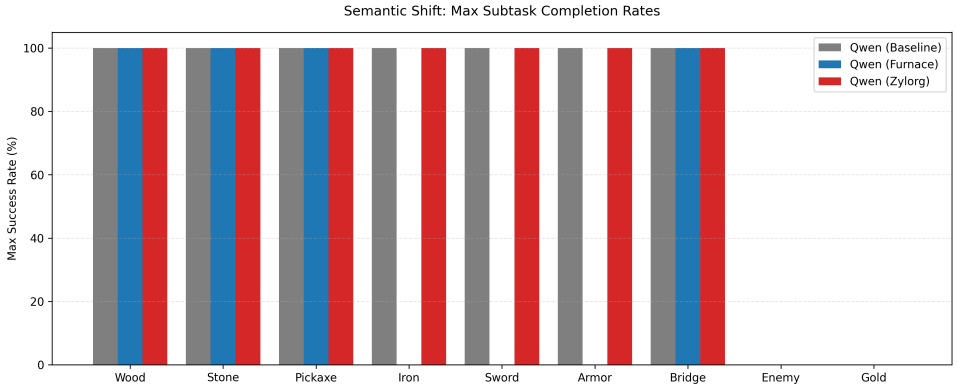


Figure 31: Maximum subtask completion rates for the Shift group. Furnace masters initial tasks (Wood, Stone, Pickaxe) but completely fails advanced tasks requiring exploration (Iron, Sword).



The results in Figures 30 and 31 illustrate the LLM's adaptive capacity, even though the `Furnace` run ultimately failed to master the final tasks.

When the Commander encountered `Zylorg`, it lacked any semantic prior. It gracefully degraded its strategy, assigning the unknown node a neutral priority weight while focusing on recognizable end-goals.

```
"reasoning": "Enemy defeat is the current bottleneck with no progress.",
"w_wood": 0.8,
"w_stone": 0.8,
"w_zylorg": 0.9,
"w_iron": 0.9,
"w_enemy": 1.2
```

Because the Commander did not artificially inflate the unknown target, the MAPPO workers organically explored the map, discovered the `Zylorg` station themselves, and solved the bottleneck. The LLM correctly stepped back and let the standard RL exploration mechanics take over. A programmed script would have simply crashed.

When the Commander encountered the `Furnace` alias, it successfully executed a zero-shot semantic mapping. It correctly reasoned that a furnace serves an identical purpose to a crafting

bench:

```
"reasoning": "Agents are not crafting Pickaxe or Sword, indicating a
bottleneck at the Workbench.",
"w_furnace": 0.4,
"w_iron": 0.2
```

The Commander successfully adapted its logic to the new terminology. The failure of the `Furnace` run was not a failure of the LLM. Rather, the LLM was *too confident*. By heavily weighting the `Furnace`, it skewed the continuous reward landscape. The MAPPO workers discovered they could accumulate massive intrinsic rewards simply by navigating to the `Furnace` and idling there. They reward-hacked the dense shaping signal and abandoned all exploration for `Iron` and `Sword`.

This experiment validates the core advantage of neuro-symbolic MARL architectures. An LLM provides immediate, code-free adaptation to new environments. However, developers must be careful: coupling an LLM's high-confidence semantic logic with continuous gradient shaping can easily trap low-level workers in local optima.

7.5 Emergent Agent Behaviors

While quantitative metrics show the LLM Commander successfully directing semantic objectives, visual analysis reveals the MAPPO workers retain their capacity for highly optimized emergent behavior.

RL agents inherently exploit mechanics to minimize execution time and maximize discounted returns. Since our environment shares inventory instantaneously, the workers discovered non-obvious cooperative strategies:

- **Asymmetric Crafting (Hit-and-Run):** Rather than gathering materials together, the policy converges on a specialized division of labor. One agent stays at the crafting station to forge weapons, while the second rushes to stand adjacent to the Enemy. Because inventory updates globally, the moment the first agent crafts the Sword, the second instantly uses it to strike. This eliminates all travel time.
- **Spawn Camping:** During the endgame, one agent often positions itself exactly on the Gold spawn tile and waits idly while its partner engages the Enemy. When the Enemy is defeated, the waiting agent collects the Gold on the very next timestep.

Although the LLM Commander provided the semantic structure, the PPO workers ruthlessly optimized the *kinetic* execution. This demonstrates a key strength of the neuro-symbolic hierarchy: the LLM prevents agents from getting lost in sparse-reward graphs, while the RL policy discovers spatial and temporal optimizations a scripted system would miss.[5]

[5] Video recordings of these emergent behaviors: `https://youtu.be/oLo0zAtuO0Y`

8 Conclusion

This thesis investigated the integration of Large Language Models (LLMs) with Multi-Agent Reinforcement Learning (MARL) to overcome exploration bottlenecks in sparse-reward environments. By structuring the environment as a strict Directed Acyclic Graph (DAG) of dependencies, we evaluated two distinct neuro-symbolic architectures: continuous Dynamic Reward Shaping (the "Mixing Board" approach) and discrete Hierarchical Reinforcement Learning (the "Meta-Controller" approach).

The results show that while LLMs can perform zero-shot semantic planning as high-level controllers for low-level kinetic workers, integrating them introduces significant challenges regarding reward stationarity, semantic grounding, and temporal alignment.

8.1 Critical Review of Findings

A primary outcome of this work was the implementation of a dynamic Potential-Based Reward Shaping (PBRS) framework. We demonstrated that an LLM could actively monitor MARL agents and shift priority weights to guide exploration, successfully unblocking mid-game bottlenecks that a standard sparse-reward MAPPO baseline failed to solve. However, we demonstrated the empirical necessity of Exponential Moving Average (EMA) smoothing; without it, discrete jumps in the Commander's weight assignments induced spikes in Temporal Difference (TD) error variance, causing temporary regressions in the underlying PPO policies and delaying convergence.

The transition to a Hierarchical Reinforcement Learning (HRL) architecture represented a decisive improvement. By allowing the Commander to dictate discrete macro-actions (Options) rather than adjusting the gravitational pull of rewards, the agents mastered the entire environment DAG, including the combat phase that the Dynamic architecture could not solve, reaching 100% completion on the final Gold milestone. Through ablation, we established that QLoRA fine-tuning is highly beneficial for sample efficiency, enabling the agents to reach the final objective approximately $5.1\times$ faster than a base model.

Despite these successes, our experiments revealed vulnerabilities in neuro-symbolic MARL paradigms, highlighting a mismatch between an LLM's semantic priors and kinetic reality. We isolated two primary failure modes:

1. We discovered that when an LLM possesses strong zero-shot knowledge of a concept (such as mapping a "Furnace" to crafting), it can confidently warp the shaping signal and trap the agents in a local optimum via reward hacking. Conversely, when presented with nonsense strings ("Zylorg"), its semantic ignorance acted as a natural regularizer, allowing the PPO exploration mechanics to solve the bottleneck.
2. Endowing the LLM with counterfactual reflection delayed convergence across all architectures:
    - For the fine-tuned model (QLoRA), reflection caused a minor delay (converging at 7.08M steps vs. 6.29M steps without reflection) because the model developed an unintended bias from its synthetic data: interpreting kinetic delays as logical failures, resulting in $2.3\times$ more staleness resets.

- For the un-finetuned base model, the failure mode was different. The base model converged in 16.32M steps without reflection but took 32.31M steps with reflection. Reflection actually *reduced* staleness resets for the base model (28,927 vs. 78,535 without reflection). This indicates that reflection prompted the base model to cycle through options rapidly enough that few ever reached the staleness threshold, but the resulting option assignments were erratic and broke environment stationarity, severely delaying PPO convergence.

These results confirm that reactive self-debugging is counterproductive if the model cannot distinguish between logical errors and physical friction.

On the practical side, routing LLM queries through cloud APIs introduced a 48% wall-clock overhead. However, our locally deployed, quantized Qwen 2.5 7B model matched the speed of a programmed baseline (7.78h vs. 8.30h), demonstrating that the computational overhead is not inherent to the architecture but specific to network-dependent deployments.

Beyond the quantitative metrics, the PPO workers also exhibited emergent cooperative strategies, such as asymmetric crafting (one agent forges while the other pre-positions near the enemy) and spawn camping (an agent waits on the Gold tile while its partner fights). These behaviors demonstrate a key strength of the neuro-symbolic hierarchy: the LLM prevents agents from getting lost in sparse-reward graphs, while the RL policy independently discovers spatial and temporal optimizations.

8.2 Prospective Vision and Recommendations

The findings of this thesis provide practical insights into the stability and failure modes of neuro-symbolic architectures, offering relevant takeaways for complex coordination problems such as robotic swarm control, automated supply chain logistics, and advanced game AI. Future research must bridge the gap between semantic logic and temporal delays.

Rather than relying on simple text-based timeout triggers, Meta-Controllers should be equipped with spatio-temporal memory or multimodal state representations. This would allow the Commander to visually or temporally differentiate between a fundamentally flawed plan and a correct plan that merely requires more physical time to execute.

Additionally, the stabilization of non-stationary objectives requires refinement. While our static KL-divergence guardrail ($\delta = 0.015$) successfully stabilized the PPO workers during option transitions, implementing an *adaptive trust region* could provide further robustness. Scaling the KL threshold dynamically based on the LLM's confidence or the specific Option type offers a more nuanced defense against objective shifts.

A promising avenue for future research is *recursive self-improvement* through RL-driven self-alignment. Recent works like STaR (Zelikman et al. 2022) and Voyager (Wang et al. 2023) demonstrate that agentic LLMs can iteratively build skill libraries or generate reasoning traces that improve their own foundation. In our MARL context, the environment serves as a ground-truth verifier: if the LLM issues an Option and the workers achieve it, that trajectory constitutes success. This trace can be distilled back into the LLM via Direct Preference Optimization (DPO), creating a closed-loop system where the LLM continuously refines its kinetic understanding based on the physical successes and failures of its agents.

Finally, while the current architecture evaluates cooperative agents executing a shared policy, a natural extension is decentralized HRL. Here, multiple LLMs could act as competing or allied faction leaders, extending the framework to adversarial or mixed-motive scenarios.

This thesis demonstrates that Large Language Models can serve as effective strategic directors for MARL agents, but future architectures must account for the temporal mismatch between semantic planning and kinetic execution.

Bibliography

ALBRECHT, Stefano V, CHRISTIANOS, Filippos et SCHÄFER, Lukas, 2024. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches* [online]. Cambridge, Massachusetts: MIT Press [visited on 2026-07-20]. Available from: https://www.marl-book.com.

AMODEI, Dario et al., 2016. Concrete Problems in AI Safety. *arXiv preprint* [online]. No. arXiv:1606.06565 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.1606.06565. arXiv:1606.06565 [cs.AI].

BELLMAN, Richard, 1957. *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press.

BENGIO, Yoshua et al., 2009. Curriculum learning. In: *Proceedings of the 26th Annual International Conference on Machine Learning* [online]. Montreal Quebec Canada: ACM, pp. 41–48 [visited on 2026-07-20]. ISBN 978-1-60558-516-1. Available from DOI: 10.1145/1553374.1553380.

CASTELLINI, Jacopo, 2022. *Improved Representations for Cooperative Multi-Agent Reinforcement Learning* [online]. [visited on 2026-07-20]. Available from: https://livrepository.liverpool.ac.uk/3158406/1/201289569_Jun2022.pdf. University of Liverpool.

CHEN, Lili et al., 2021. Decision Transformer: Reinforcement Learning via Sequence Modeling. *arXiv preprint*.

CHUNG, Junyoung et al., 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. No. arXiv:1412.3555. Available from DOI: 10.48550/arXiv.1412.3555. arXiv:1412.3555 [cs.NE].

DETTMERS, Tim et al., 2022. *bitsandbytes: 8-bit optimizers, matrix multiplication, and quantization* [online]. [visited on 2026-07-20]. Available from: https://github.com/TimDettmers/bitsandbytes.

DETTMERS, Tim et al., 2023. QLoRA: Efficient Finetuning of Quantized LLMs. *arXiv preprint* [online]. No. arXiv:2305.14314 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.2305.14314. arXiv:2305.14314 [cs].

FOERSTER, Jakob et al., 2017. Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning. *arXiv preprint* [online]. No. arXiv:1702.08887 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.1702.08887. arXiv:1702.08887 [cs].

HARRIS, Charles R., MILLMAN, K. Jarrod, WALT, Stéfan J. van der, et al., 2024. *NumPy* [online]. Version 1.26.4 [visited on 2026-07-20]. Available from: https://numpy.org/.

HAUSKNECHT, Matthew et STONE, Peter, 2015. Deep Recurrent Q-Learning for Partially Observable MDPs. *arXiv preprint* [online]. No. arXiv:1507.06527 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.1507.06527. arXiv:1507.06527 [cs].

HOCHREITER, Sepp et SCHMIDHUBER, Jürgen, 1997. Long Short-Term Memory. *Neural Computation*. Vol. 9, pp. 1735–1780. Available from DOI: 10.1162/neco.1997.9.8.1735.

HU, Edward J. et al., 2021. LoRA: Low-Rank Adaptation of Large Language Models. *arXiv preprint* [online]. No. arXiv:2106.09685 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.2106.09685. arXiv:2106.09685 [cs.CL].

KATSIKOPOULOS, K.V. et ENGELBRECHT, S.E., 2003. Markov decision processes with delays and asynchronous cost collection. *IEEE Transactions on Automatic Control*. Vol. 48, no. 4, pp. 568–574. ISSN 1558-2523. Available from DOI: 10.1109/TAC.2003.809799.

KULKARNI, Tejas D. et al., 2016. Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation. *arXiv preprint* [online]. No. arXiv:1604.06057 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.1604.06057. arXiv:1604.06057 [cs.LG].

LECUN, Y. et al., 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*. Vol. 86, no. 11, pp. 2278–2324. ISSN 1558-2256. Available from DOI: 10.1109/5.726791.

LINIETSKY, Juan, MANZUR, Ariel, et al., 2023. *Godot Engine* [online]. Version 4.2.1 [visited on 2026-07-20]. Available from: https://godotengine.org/.

LOWE, Ryan et al., 2017. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. *arXiv preprint* [online]. No. arXiv:1706.02275 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.1706.02275. arXiv:1706.02275 [cs].

MA, Yecheng Jason et al., 2024. Eureka: Human-Level Reward Design via Coding Large Language Models. *arXiv preprint arXiv:2310.12931* [online] [visited on 2026-07-20]. Available from: https://arxiv.org/abs/2310.12931.

MANGRULKAR, Sourab et al., 2022. *PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods* [online]. [visited on 2026-07-20]. Available from: https://github.com/huggingface/peft.

MITCHENER, Ludovico et al., 2022. Detect, Understand, Act: A Neuro-symbolic Hierarchical Reinforcement Learning Framework. *Machine Learning*. Vol. 111, no. 4, pp. 1523–1549. ISSN 1573-0565. Available from DOI: 10.1007/s10994-022-06142-7.

NG, Andrew Y., HARADA, Daishi et RUSSELL, Stuart J., 1999. Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping. In: *Proceedings of the Sixteenth International Conference on Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 278–287. ICML'99. ISBN 978-1-55860-612-8.

OLIEHOEK, Frans A. et AMATO, Christopher, 2016. *A Concise Introduction to Decentralized POMDPs* [online]. Cham: Springer International Publishing [visited on 2026-07-20]. SpringerBriefs in Intelligent Systems. ISBN 978-3-319-28927-4. Available from DOI: 10.1007/978-3-319-28929-8.

OLLAMA TEAM, 2024. *Ollama Framework for Local Large Language Models*. Available also from: https://ollama.com/.

PASZKE, Adam et al., 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv preprint* [online]. No. arXiv:1912.01703 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.1912.01703. arXiv:1912.01703 [cs.LG].

PETRENKO, Aleksei et al., 2020. Sample Factory: Egocentric 3D Control from Pixels at 100000 FPS with Asynchronous Reinforcement Learning. In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. PMLR. Vol. 119, pp. 7652–7662. Proceedings of Machine Learning Research. Available also from: http://proceedings.mlr.press/v119/petrenko20a.html.

PORTELAS, Rémy et al., 2020. Automatic Curriculum Learning For Deep RL: A Short Survey. No. arXiv:2003.04664. Available from DOI: 10.48550/arXiv.2003.04664. arXiv:2003.04664 [cs.LG].

QWEN et al., 2024. Qwen2.5 Technical Report. No. arXiv:2412.15115. Available from DOI: 10.48550/arXiv.2412.15115. arXiv:2412.15115 [cs.CL].

RADFORD, Alec et al., 2018. Improving Language Understanding by Generative Pre-Training. *OpenAI* [online] [visited on 2026-07-20]. Available from: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

RASHID, Tabish et al., 2018. QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning. *arXiv preprint* [online]. No. arXiv:1803.11485 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.1803.11485. arXiv:1803.11485 [cs].

SCHULMAN, John et al., 2017. Proximal Policy Optimization Algorithms. *arXiv preprint* [online]. No. arXiv:1707.06347 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.1707.06347. arXiv:1707.06347 [cs.LG].

SCHWARTZ, Roy et al., 2020. Green AI. *Communications of the ACM*. Vol. 63, no. 12, pp. 54–63. Available from DOI: 10.1145/3381831.

SHINN, Noah et al., 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. *arXiv preprint* [online]. No. arXiv:2303.11366 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.2303.11366. arXiv:2303.11366 [cs.AI].

SHINNERS, Pete et al., 2023. *Pygame* [online]. Version 2.5.2 [visited on 2026-07-20]. Available from: https://www.pygame.org/.

STRUBELL, Emma, GANESH, Ananya et MCCALLUM, Andrew, 2019. Energy and Policy Considerations for Deep Learning in NLP. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 3645–3650.

SUNEHAG, Peter et al., 2017. Value-Decomposition Networks For Cooperative Multi-Agent Learning. *arXiv preprint* [online] [visited on 2026-07-20]. Available from: https://arxiv.org/abs/1706.05296v1.

SUTTON, Richard S. et BARTO, Andrew, 2018. *Reinforcement learning: an introduction*. Second edition. Cambridge, Massachusetts London, England: The MIT Press. Adaptive computation and machine learning. ISBN 978-0-262-03924-6.

SUTTON, Richard S., PRECUP, Doina et SINGH, Satinder, 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*. Vol. 112, no. 1, pp. 181–211. ISSN 0004-3702. Available from DOI: 10.1016/S0004-3702(99)00052-1.

SZEPESVÁRI, Csaba, 2011. Reinforcement Learning Algorithms for MDPs. In: *Wiley Encyclopedia of Operations Research and Management Science* [online]. 1st ed. Wiley [visited on 2026-07-20]. ISBN 978-0-470-40063-0. Available from DOI: 10.1002/9780470400531.eorms0714.

TERRY, J. K. et al., 2021. PettingZoo: Gym for Multi-Agent Reinforcement Learning. *arXiv preprint* [online]. No. arXiv:2009.14471 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.2009.14471. arXiv:2009.14471 [cs.LG].

VASWANI, Ashish et al., 2017. Attention Is All You Need. *arXiv preprint* [online]. No. arXiv:1706.03762 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.1706.03762. arXiv:1706.03762 [cs.CL].

WANG, Guanzhi et al., 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. *arXiv preprint arXiv:2305.16291*.

WEI, Jason et al., 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *arXiv preprint* [online]. No. arXiv:2201.11903 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.2201.11903. arXiv:2201.11903 [cs.CL].

WERRA, Leandro von et al., 2020. *TRL: Transformer Reinforcement Learning* [online]. [visited on 2026-07-20]. Available from: https://github.com/huggingface/trl.

WOLF, Thomas et al., 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *arXiv preprint* [online]. No. arXiv:1910.03771 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.1910.03771. arXiv:1910.03771 [cs.CL].

YAN, Jingyuan et al., 2026. GLARE: Scalable Neuro-Symbolic Reward Shaping for LLM Agents via Group-Level Automata. In: *Forty-third International Conference on Machine Learning* [online] [visited on 2026-07-20]. Available from: https://openreview.net/forum?id=3a8fm24EQd.

YANG, An et al., 2025. Qwen3 Technical Report. *arXiv preprint* [online]. No. arXiv:2505.09388 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.2505.09388. arXiv:2505.09388 [cs.CL].

YAO, Shunyu et al., 2023. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629* [online] [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.2210.03629.

YU, Chao et al., 2022. The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games. *arXiv preprint arXiv:2103.01955* [online] [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.2103.01955.

ZELIKMAN, Eric et al., 2022. STaR: Bootstrapping Reasoning With Reasoning. *Advances in Neural Information Processing Systems*. Vol. 35, pp. 15476–15488.

ZHANG, Xiuhui et al., 2026. Progress Reward Model for Reinforcement Learning via Large Language Models. *arXiv preprint* [online] [visited on 2026-07-20]. Available from: https://github.com/deng-ai-lab/PRM4RL.

ZHU, Xizhou et al., 2023. Ghost in the Minecraft: Generally Capable Agents for Open-World Environments via Large Language Models with Text-based Knowledge and Memory. *arXiv preprint* [online]. No. arXiv:2305.17144 [visited on 2026-07-20]. Available from DOI: 10.48550/arXiv.2305.17144. arXiv:2305.17144 [cs.AI].

Use of artificial intelligence-assisted tools

In the context of this work, the author states that he used artificial intelligence-assisted tools for the following purposes:

- **Formal Improvements**: Assisted with spelling, syntax, phrasing, and report structuring throughout all chapters.
  AI tools used: Gemini 3.1 Pro (via Antigravity Assistant)
- **Substantive Reflection**: Served as a collaborative sounding board for deep reflection during architectural design, debugging, and data interpretation across Chapters 4, 5, 6, and 7. The author remained the principal instigator and decision-maker for all methodologies and conclusions.
  AI tools used: Gemini 3.1 Pro (via Antigravity Assistant)
- **Asset Generation**: Assisted in generating 2D pixel-art sprites and visual assets used in the custom MARL environment renderer (Chapter 4).
  AI tools used: Gemini 1.5 Pro

Appendix 1: Core Algorithm Implementations

Listing 1: DAG guardrail enforcement in `orchestrator.py`.

```python
def apply_dag_guardrails(weights, metrics):
    guarded = dict(weights)
    # Iron requires Pickaxe
    if metrics.get('pickaxe', 0.0) < 1.0:
        guarded['w_iron'] = 0.0
    # Enemy and Gold require Bridge+Sword+Armor
    if (metrics.get('bridge', 0.0) < 1.0
        or metrics.get('sword', 0.0) < 1.0
        or metrics.get('armor', 0.0) < 1.0):
        guarded['w_enemy'] = 0.0
        guarded['w_gold'] = 0.0
    return guarded
```

Listing 2: Vectorised PBRS computation across all environments in `orchestrator.py`.

```python
# Calculate L2 Norm distances for all agents simultaneously
dist_next = np.linalg.norm(pos_next - goal_targets, axis=2)
dist_prev = np.linalg.norm(pos_prev - goal_targets, axis=2)

MAX_DIST = 85.0
phi_next = MAX_DIST - dist_next
phi_prev = MAX_DIST - dist_prev

# Apply gamma discount, scaling, and the active mask
F = (gamma * phi_next - phi_prev) * R_LLM_SCALE * goal_active
```

Listing 3: Per-agent intrinsic shaping in `hrl_train_loop.py`.

```python
z0 = get_option_target_zone(a0_opt)
idle_mask_0 = (z0 == -1)
safe_z0 = np.where(idle_mask_0, 0, z0)

dist0_prev = np.linalg.norm(
    pos_prev[:, 0] - vec_env.zones[np.arange(n_envs), safe_z0],
    axis=1)
dist0_next = np.linalg.norm(
    vec_env.pos[:, 0] - vec_env.zones[np.arange(n_envs), safe_z0],
    axis=1)
phi0_prev = MAX_DIST - dist0_prev
phi0_next = MAX_DIST - dist0_next
shaping0 = np.where(
    idle_mask_0, 0.0,
    (0.99 * phi0_next) - phi0_prev
)
intrinsic_r[:, 0] = np.where(
    a0_success, 1.0, shaping0 * 0.05
)
```

Listing 4: Composite reward in `hrl_train_loop.py`.

```
total_r = env_rewards + 0.5 * intrinsic_r
```

Listing 5: Target KL guard in `hrl_train_loop.py`.

```
with torch.no_grad():
    approx_kl = (-logratio).mean().item()
target_kl = 0.015
if approx_kl > target_kl:
    target_kl_reached = True
    break
```