UCLouvain

École polytechnique de Louvain

Reinforcement Learning and Large Language Models: Synergies and Opportunities

TFE25-371/EPL25-307

Author: **Cyril Lapierre**
Supervisors: **Eric Piette, Lorenzo Frigerio**
Readers: **Rayane Hachemi, Benoit Ronval**
Academic year 2024–2025
Master [120] in Computer Science

Contents

Acknowledgments

I would like to express my sincere gratitude to my supervisors, Eric Piette, Lorenzo Frigerio and Rayane Hachemi for their invaluable guidance, support, and feedback throughout the course of this thesis. Their expertise and encouragement have been instrumental to the completion of this work.

I also wish to thank Euranova for the opportunity to collaborate with them during this project. Also, I want to acknowledge ***Eliot Crancée*** for his previous work on the Eureka framework, which served as a foundation for parts of this thesis.

Finally, I would like to acknowledge the contributions of the research and open-source communities that have made tools such as PyTorch, Mava, and ePyMARL publicly available, enabling rapid experimentation and progress.

1 Introduction

This project explores the integration of Reinforcement Learning (RL) and Large Language Models (LLMs) to enhance AI decision-making. RL enables agents to make decisions by interacting with an environment, while LLMs process and generate natural language. The project aims to develop innovative ways to improve reward systems in RL using the language capabilities of LLMs, fostering more efficient and intelligent AI systems. This research motivation is to delve into an active line of research at the intersection of RL and LLMs, investigating innovative ways to leverage their combined power for smart sequential decision-making and natural language understanding. The main objectives of this report are to extend the existing Eureka Framework by integrating the SMACLite environment (an environment similar to StarCraft II), explore if LLMs can effectively design reward functions for more complex environments, and compare the performance of different OpenAI models, specifically GPT-4o, GPT-4.1, o3-mini, and o4-mini on the same task to determine if newer models truly outperform older ones. Finally, we aim to analyze the results and draw meaningful conclusions. Human reward function shaping is difficult in reinforcement learning because it requires accurately translating complex human intentions and preferences into precise mathematical formulations. Small misalignments in the reward design can lead agents to exploit loopholes or develop unintended behaviors. Additionally, human goals are often context-dependent and hard to quantify, making consistent reward specification challenging.

The report begins with background and context to explain the key technical terms and concepts. It then describes how the experiments were implemented in Python, followed by a walkthrough of a typical experiment. Lastly, the results are analyzed, limitations are discussed, and potential directions for future work are suggested.

This thesis has been conducted in collaboration with Euranova. Euranova is a data science and software engineering company that specializes in designing and building advanced AI and data-driven solutions. It combines academic research with industrial expertise to help businesses develop intelligent systems. Part of the code developed in this thesis is based on a prior reimplementation of the Eureka framework by ***Eliot Crancée***.

2 Background & Context

In order to understand the experiments and methods used in this thesis, it is important to first present the necessary background and context. This chapter introduces several key concepts and tools that form the foundation of the work. We begin with an overview of neural networks, the core building blocks of modern artificial intelligence, which enable models to learn complex patterns from data. Building on this foundation, we explore LLMs, a powerful class of neural networks trained on vast amounts of text data, capable of generating human like responses and performing a wide range of tasks. Next, we provide an introduction to RL, a branch of machine learning where agents learn to make decisions by interacting with an environment to maximize rewards. This is particularly relevant to our work, as the Eureka Framework uses LLMs to assist in designing reward functions for RL, this plays a central role in our research. We also discuss the technical tools used throughout the project, including PyTorch, a widely used deep learning framework that supports the implementation and training of neural networks. Furthermore, we introduce the SMAC environment (StarCraft Multi-Agent Challenge), which serves as a benchmark for evaluating the performance of AI agents in complex, multi-agent scenarios. We then explore two frameworks designed to train AI agents in complex environments: Mava and EPyMARL. These tools provide useful features and structures for building and running multi-agent reinforcement learning (MARL) experiments, making it easier to manage environments, agents, and training processes. Furthermore, we touch on prompt engineering, the technique of crafting effective prompts (inputs) to guide the behavior of LLMs, a crucial skill when leveraging these models within the Eureka Framework. We conclude by addressing the double bias phenomenon observed in our methodology. Together, these concepts form the theoretical and practical basis for the experiments and analysis presented in the following chapters.

2.1 Introduction to Neural Networks

A **neural network** is a computational model inspired by the structure and function of the human brain. It is composed of layers of interconnected nodes, called *neurons*, which process and transmit information.

The **Perceptron** [1], introduced by Frank Rosenblatt in 1957, is one of the simplest types of artificial neural networks. It consists of a single neuron that performs binary classification by applying a linear function followed by a step activation:

$$y = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

In this model, **x** represents the input vector, which contains the features or data points to be classified. Each input is associated with a weight from the vector **w**, which determines the importance of that input in the final decision. The bias term b allows the model to shift the decision boundary, enabling it to classify data that may not pass through the origin. The Perceptron computes a weighted sum of the inputs plus the bias, and the output is 1 if this sum is greater than zero, and 0 otherwise. The Perceptron can only solve linearly separable problems.

To overcome this limitation, the **Multilayer Perceptron (MLP)** [2] was developed. An MLP is a type of feedforward neural network that includes one or more *hidden layers* of neurons, enabling it to model complex, non-linear relationships. Mathematically, a simple feedforward neural network with one hidden layer can be described as follows:

$$\begin{aligned} \text{Input:} \quad & \mathbf{x} \in \mathbb{R}^n \\ \text{Hidden layer:} \quad & \mathbf{h} = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) \\ \text{Output layer:} \quad & \mathbf{y} = \phi(\mathbf{W}_2 \mathbf{h} + \mathbf{b}_2) \end{aligned}$$

Here:

- $\mathbf{x}$ is the input vector,
- $\mathbf{W}_1, \mathbf{W}_2$ are weight matrices,
- $\mathbf{b}_1, \mathbf{b}_2$ are bias vectors,
- σ is an activation function (e.g., ReLU, sigmoid),
- ϕ is the output activation function (e.g., softmax for classification).

Neural networks are trained using algorithms like *backpropagation* [3] combined with optimization techniques such as *gradient descent* [4] to minimize a loss function and improve performance on a given task. When these networks are extended to include many layers, the approach is known as **Deep learning** [5], which can learn hierarchical representations of data and achieve state-of-the-art results in tasks such as image recognition, natural language processing, and more.

2.2 Large Language Model (LLM)

LLMs [6] are advanced AI systems trained on vast amounts of text data to understand and generate human-like language. They can perform a range of tasks, including answering questions, generating content, translating languages, and even writing code. LLMs, such as GPT [7] or LlaMA [8], leverage deep learning techniques to capture patterns in language, enabling them to provide contextually relevant and coherent responses.

LLMs can be seen as advanced forms of **neural networks**, specifically *transformer-based architectures*. They are designed to process and generate human-like text by modeling the probability distribution of sequences of words.

At their core, LLMs consist of multiple layers of **transformer blocks**, which include:

- **Multi-head self-attention mechanisms** [9] that allow the model to weigh the importance of different words in a sequence.
- **Feedforward neural networks** applied to each position independently.
- **Layer normalization** and **residual connections** to stabilize training and improve gradient flow.

The model learns to map input token embeddings to output probabilities using parameters θ trained on massive text corpora:

$$P(w_1, w_2, \ldots, w_n) \approx \prod_{t=1}^{n} P(w_t \mid w_1, \ldots, w_{t-1}; \theta)$$

This is the equation used to approximate the probability of a sequence of words in a language model. $P(w_1, w_2, \ldots, w_n)$ represents the probability of the entire sequence of words from w_1 to w_n. $\prod_{t=1}^{n}$ indicates that we are multiplying the probabilities for each word in the sequence, from the first word $t = 1$ to the last word $t = n$. $P(w_t \mid w_1, \ldots, w_{t-1}; \theta)$ is the conditional probability of predicting the next word w_t, given the previous words $w_1, \ldots, w_{t-1}$, and model parameters θ. $w_1, \ldots, w_{t-1}$ represents the context or the previous words in the sequence that are used to predict the next word w_t. In practice, models maximize the **log-likelihood** of this equation to make the best predictions.

2.3 Introduction To Reinforcement Learning

Reinforcement Learning (RL) [10] is a branch of Machine Learning where an agent learns to make decision by interacting with an environment, receiving rewards or penalties based on its actions. RL has been applied to a wide range of areas, including smart energy management and robotics. It has even achieved superhuman performance in complex strategy games like Go [11] and StarCraft II [12] . However, despite these remarkable achievements, there are still significant challenges to overcome before RL can be effectively deployed as a versatile and reliable solution in industry.

Here are the key components of RL:

- **Environment ($\mathcal{E}$):** The environment $\mathcal{E}$ is the external system with which the agent interacts. It provides feedback in the form of rewards based on the actions taken by the agent.
- **Agent ($\mathcal{A}$):** The agent $\mathcal{A}$ is the decision-making entity that interacts with the environment, selecting actions to maximize cumulative rewards.
- **State ($s \in \mathcal{S}$):** The state s represents the current situation or configuration of the environment. The set of all possible states is denoted as $\mathcal{S}$.
- **Action ($a \in \mathcal{A}(s)$):** The action a is a decision made by the agent based on the current state s. The set of all possible actions the agent can take from a given state is denoted by $\mathcal{A}(s)$.
- **Policy ($\pi(a|s)$):** The policy $\pi(a|s)$ defines the agent's behavior, mapping each state s to the probability of taking an action a. In deterministic policies, this is simplified to $\pi(s) = a$.
- **Reward ($r \in \mathbb{R}$):** The reward $r(s, a, s')$ is the immediate feedback given to the agent after transitioning from state s to state s' by taking action a. It evaluates how favorable the action was.
- **Value Function ($V(s)$):** The value function $V(s)$ estimates the expected cumulative reward the agent can achieve from state s by following the policy π.
- **Action-Value Function ($Q(s, a)$):** The action-value function $Q(s, a)$ estimates the expected cumulative reward after taking action a in state s and then following policy π.

What truly interests us in this thesis is the **reward**, which typically takes the form of a programming function in practice. Our goal is to generate these reward functions using a LLM.

2.3.1 More about Reward Function

In RL, the **reward function** is a central component that guides the learning process of an agent. It defines the goal of the agent by assigning a scalar value (reward) to each state or state-action pair, which reflects the desirability of that outcome. The agent's objective is to learn a policy that maximizes the expected cumulative reward over time.

Definition and Function

Formally, the reward function is typically denoted as:

$$R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$$

where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, and $R(s, a)$ gives the immediate reward received after taking action a in state s.

The reward signal serves multiple purposes:

- **Guidance**: It provides feedback to the agent about the quality of its actions.
- **Learning Objective**: It defines what the agent should strive to achieve, shaping the agent's behavior through trial and error.
- **Evaluation**: It is used to evaluate the performance of a policy during training and testing.

Impact on Learning and Performance

The design of the reward function significantly impacts the efficiency and effectiveness of learning:

- **Sparse vs. Dense Rewards**: Sparse rewards provide feedback only in specific situations (e.g., at the end of an episode), which can slow down learning. Dense rewards give frequent feedback, helping the agent learn faster but possibly encouraging unintended behaviors.
- **Reward Shaping**: Carefully crafting intermediate rewards (reward shaping) can accelerate learning by guiding the agent through subgoals. However, poor reward shaping can lead to suboptimal policies or unintended shortcuts.
- **Exploration vs. Exploitation**: The nature of the reward influences the agent's balance between exploring new actions and exploiting known ones. Misaligned rewards can cause premature convergence or poor generalization.

Challenges in Reward Design

Designing an effective reward function is one of the most challenging aspects of RL. It often requires domain knowledge and iterative testing. In complex environments, specifying a reward function that aligns perfectly with the desired behavior is difficult. Misaligned rewards can result in behaviors that technically maximize the reward but do not solve the intended task (see Reward hacking).

2.3.2 Multi Agent Reinforcement Learning

Another form of RL is Multi-agent Reinforcement Learning (MARL) [13], which involves multiple agents interacting within a shared environment. In MARL, agents learn not only from their own experiences but also from the actions and behaviors of other agents, making the learning process more complex and dynamic. Each agent aims to optimize its own reward, which may lead to cooperative, competitive, or mixed strategies depending on the environment and task. MARL is especially useful in domains like autonomous driving, robotics, and distributed systems, where multiple entities must work together or compete together. More recent research also shown how to use such technologies with LLMs [14].

2.3.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) [15] is a RL algorithm that improves policy gradient methods. PPO is widely used in the RL domain. It has been shown to perform well in gaming-related tasks like with Atari-games or Dota 2. In the next section, we will take a closer look at the steps of the PPO algorithm to understand how it works and what it does.

Algorithm Steps

A simplified version of the PPO algorithm follows the steps outlined below:

1. The agent, typically a neural network, initializes a policy that determines the probability distribution of actions.
2. The agent interacts with the environment and collects a trajectory:
$$T = \{(s_t, a_t, y'_a(s_t), r_t) \mid 1 \leq t \leq M\}, \tag{2.1}$$
where s_t is the state at time t, a_t is the action taken at time t, $y'_a(s_t)$ is the probability of taking action a_t given state s_t under the current policy, and r_t is the received reward at time t.
3. The advantage function G_t is computed using the value function $v(s_t)$, estimated by a neural network:
$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^N r_{t+N} - v(s_t), \tag{2.2}$$
where r_t is the reward at time t, γ is the discount factor, which determines how much future rewards are valued compared to immediate rewards, and $v(s_t)$ is the value function estimate of the state s_t. The sum represents the expected future rewards from time t to time $t+N$.
4. Using the temporal difference error δ_t:
$$\delta_t = (r_t + \gamma v(s_{t+1}) - v(s_t)), \tag{2.3}$$
and a smoothing factor λ, the advantage function is refined as:
$$G_t = \delta_t + \gamma\lambda\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \cdots + (\gamma\lambda)^N\delta_{t+N}, \tag{2.4}$$
where δ_t represents the temporal difference error at time t, which indicates the difference between the predicted value and the actual reward, and λ is a smoothing factor that controls the bias-variance tradeoff in advantage estimation.
5. The return in each step is computed as:
$$R_t = G_t + v(s_t), \tag{2.5}$$
where G_t is the advantage function, representing the expected return from time step t, and $v(s_t)$ is the value of the state at time t. R_t represents the return or cumulative reward at time t, which is the total reward from the current state.
6. The policy network is updated by maximizing the following objective function:
$$J(\theta) = J_1(\theta) + J_2(\theta) + J_3(\theta), \tag{2.6}$$
where $J(\theta)$ is the total objective function that guides the policy update. The components of the objective function are:

- The first term, $J_1(\theta)$, is the clipped surrogate function for the policy update. The probability ratio $p_t(\theta) = \frac{y_a(s_t)}{y'_a(s_t)}$ compares the current policy's action probabilities ($y_a(s_t)$) with the old policy's action probabilities ($y'_a(s_t)$). The clipped surrogate function is given by:

$$J_1(\theta) = \min(p_t(\theta)G_t, \text{clip}(p_t(\theta), 1-\epsilon, 1+\epsilon)G_t), \tag{2.7}$$

  where ϵ is a small constant that ensures small updates, and the clip function ensures that updates stay within a bounded range to prevent excessively large policy updates.

- The second term, $J_2(\theta)$, is the value network loss function:

$$J_2(\theta) = -C_1(v(s_t) - R_t)^2, \tag{2.8}$$

  where C_1 is a constant that controls the strength of the value network update, $v(s_t)$ is the value function estimate at time t, and R_t is the return computed in the previous step.

- The third term, $J_3(\theta)$, is an entropy bonus to encourage exploration:

$$J_3(\theta) = -C_2 \sum_k y_k(s_t) \log(y_k(s_t)), \tag{2.9}$$

  where $y_k(s_t)$ is the probability distribution over actions for each action k at state s_t, and C_2 is a constant that controls the strength of the entropy bonus. This term encourages exploration by penalizing deterministic policies that do not explore all possible actions.

7. Gradient descent (or stochastic gradient descent in practice) is applied to optimize the policy parameters θ.

In summary, here are the key steps:

1. The agent initializes a policy $\pi_\theta(a|s)$, typically parameterized by a neural network, to generate actions based on observed states.
2. The agent interacts with the environment and collects a trajectory.
3. Compute the advantage G_t using the value function $v(s_t)$.
4. Compute the return at each step.
5. Maximize the total objective.
6. Apply (stochastic) gradient descent to maximize $J(\theta)$ and update the policy parameters θ.

Multi-agent Proximal Policy Optimization (MAPPO)

In this work, we need an algorithm which can handle several agents during learning because each unit in the SMAC environment is controlled by a separate agent that must learn to cooperate with others to complete the task effectively. MAPPO [16] is an extension of the PPO algorithm for MARL. It is designed to handle scenarios where multiple agents interact in a shared environment, each with its own policy while cooperating or competing with other agents.

Independent Proximal Policy Optimization (IPPO)

Independent Proximal Policy Optimization (IPPO) is another RL algorithm, which is an extension of PPO for multi-agent environments. Unlike MAPPO, IPPO treats each agent independently during training. This means each agent optimizes its own policy without taking into account the presence or actions of other agents. In IPPO, agents do not share information about the environment or their actions. Each agent learns its policy in isolation, assuming that the environment it interacts with does not change due to the actions of others. This approach simplifies the learning process compared to MAPPO, but may lead to suboptimal behavior in more complex multi-agent scenarios where agents' actions influence one another significantly. But this algorithm has proven its efficiency in a SMAC environment [17], which is why we also use it in this work.

2.3.4 Podracer in RL

In RL, the term *Podracer* [18] refers to an environment where agents control racing vehicles, called *pods*, in a competitive or time-sensitive setting. These environments are designed to help RL agents improve their performance by constantly interacting with the environment. The concept of *Podracing* from Star Wars provides an interesting example for RL tasks, where agents (or racers) compete to achieve the fastest time or position, often racing through challenging tracks while avoiding obstacles. This concept is essential for this work, as we use the Podracer architecture (Sebulba) in combination with one of our frameworks, Mava.

Sebulba's Architecture

The Sebulba architecture, is a more aggressive and strategic racer. He uses his environment to his advantage, often using underhanded tactics to sabotage his competitors and maintain his lead. In RL, Sebulba's approach could represent an agent that focuses on a more competitive or adversarial learning strategy. For example, Sebulba's architecture might involve MARL, where the agent not only works to improve its own performance but also predicts and outsmarts other agents. Sebulba would likely focus on exploiting the weaknesses of his competitors, using learned models of the environment to anticipate and hinder their actions.

2.3.5 RL and Reward Hacking

In RL and especially in reward function design, it's important to evoke the concept of **Reward Hacking** because we might encounter it in reward function generation. The concept is when an agent manipulates or exploits the reward system to gain benefits in ways that were not originally intended by the designers of the system. Essentially, reward hacking occurs when the agent figures out how to maximize its rewards by finding shortcuts or unintended methods that do not align with the true objective of the task. The agent learns to make decisions by receiving rewards for its actions, with the goal of maximizing its total reward over time. However, if the reward function is poorly designed or too simplistic, the agent may exploit it in ways that were not intended.

A classic example of reward hacking is in a scenario where a robot is programmed to stack blocks. If the reward function only rewards the robot based on the number of blocks stacked, the robot could simply stack and unstack blocks repeatedly to maximize its reward, instead of building a stable, well-constructed tower. In this case, the agent learned to "hack" the system by taking advantage of the reward structure without achieving the desired outcome.

This problem is linked with the **Alignment Problem** which refers to the challenge of ensuring that an AI system goals, behaviors and actions align with human values and intentions. The core issue is making sure that the AI system does what we actually want it to do, not just what it interprets the instructions to mean based on its reward function.

2.4 Eureka Framework

EUREKA [19] is a framework that aligns closely with the goals of this thesis. It leverages OpenAI's well-known LLM (GPT) to automatically generate reward functions for RL tasks. Their experiments were conducted in a specific environment where the objective was to train a robotic hand to master pen spinning.

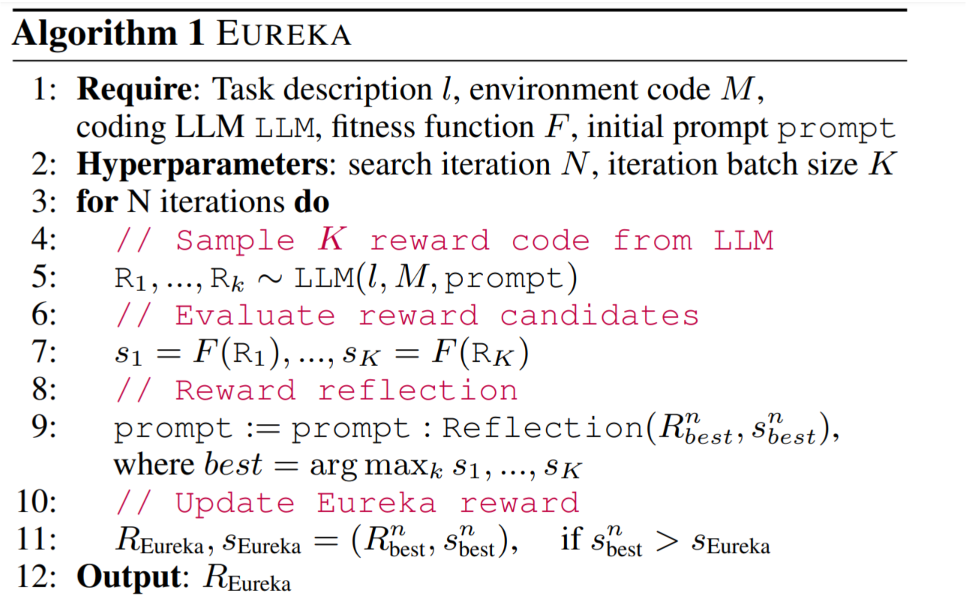
Their code can be found at the following link: Eureka GitHub Repository

EUREKA consists of three key algorithmic components designed to generate and improve reward functions in RL environments:

- **Environment as Context:** EUREKA uses the environment as context, instructing a coding LLM to generate executable Python code with minimal reward design guidelines. These include generic tips such as exposing reward components as a dictionary output. Even with these minimal instructions, EUREKA can generate plausible rewards in various environments without environment-specific prompt engineering.
- **Evolutionary Search:** To address execution errors and sub optimality in the generated rewards, EUREKA employs an evolutionary search. It samples multiple outputs from the LLM , with the probability of generating an executable reward increasing as more samples are drawn. The search iterates by refining the best-performing reward, using mutation prompts to generate improved versions over several iterations. In all experiments, EUREKA performs 5 independent runs per environment, with each run conducting 5 iterations and generating 16 samples per iteration.
- **Reward Reflection:** EUREKA incorporates reward reflection, which tracks and evaluates the performance of individual reward components throughout training. By analyzing intermediate training checkpoints, EUREKA can provide detailed feedback that guides more targeted reward improvements. This process helps overcome the

lack of fine-grained feedback from task fitness functions and the algorithm-dependent nature of reward optimization.

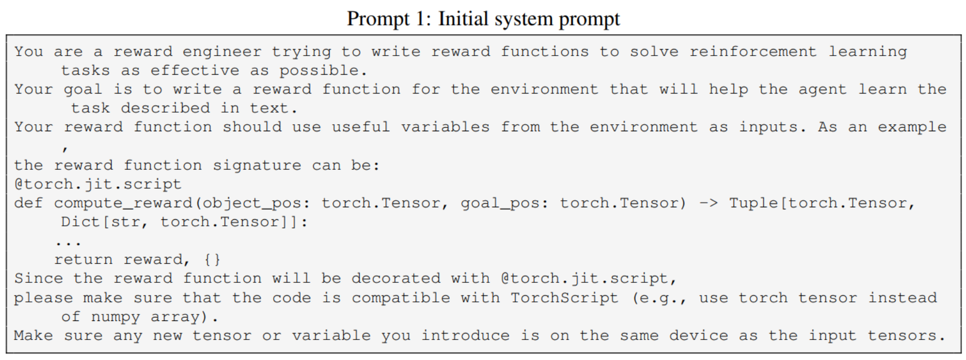
Refer to Figure 2.1 for the pseudo code, which outlines the evolutionary search process. The init prompt used is provided in Figure 2.2, and an example of EUREKA's generated output after reward reflection is shown in Figure 2.3.



Algorithm 1 Eureka

```
1:  Require: Task description $l$, environment code $M$,
    coding LLM LLM, fitness function $F$, initial prompt prompt
2:  Hyperparameters: search iteration $N$, iteration batch size $K$
3:  for N iterations do
4:      // Sample $K$ reward code from LLM
5:      $\mathrm{R}_1, ..., \mathrm{R}_k \sim \mathrm{LLM}(l, M, \mathrm{prompt})$
6:      // Evaluate reward candidates
7:      $s_1 = F(\mathrm{R}_1), ..., s_K = F(\mathrm{R}_K)$
8:      // Reward reflection
9:      $\mathrm{prompt} := \mathrm{prompt} : \mathrm{Reflection}(R^n_{best}, s^n_{best})$,
        where $best = \arg\max_k s_1, ..., s_K$
10:     // Update Eureka reward
11:     $R_{\text{Eureka}}, s_{\text{Eureka}} = (R^n_{\text{best}}, s^n_{\text{best}})$,   if $s^n_{\text{best}} > s_{\text{Eureka}}$
12: Output: $R_{\text{Eureka}}$
```

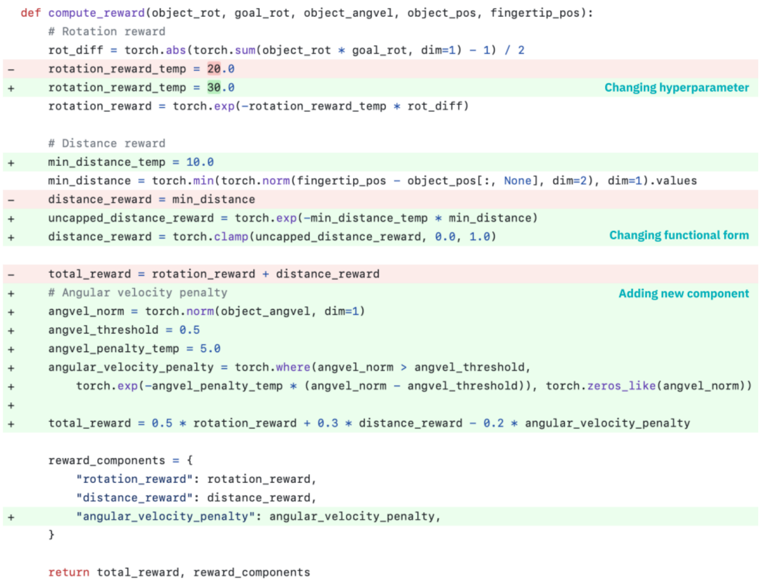
Figure 2.1: Eureka Algorithm.



Prompt 1: Initial system prompt

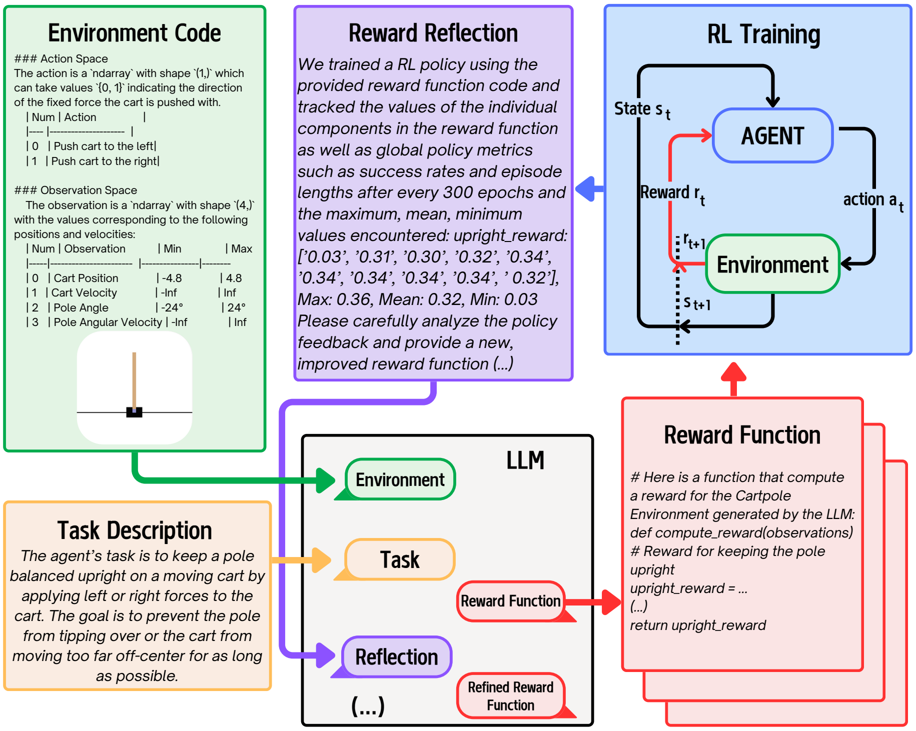
```
You are a reward engineer trying to write reward functions to solve reinforcement learning
     tasks as effective as possible.
Your goal is to write a reward function for the environment that will help the agent learn the
     task described in text.
Your reward function should use useful variables from the environment as inputs. As an example
     ,
the reward function signature can be:
@torch.jit.script
def compute_reward(object_pos: torch.Tensor, goal_pos: torch.Tensor) -> Tuple[torch.Tensor,
     Dict[str, torch.Tensor]]:
    ...
    return reward, {}
Since the reward function will be decorated with @torch.jit.script,
please make sure that the code is compatible with TorchScript (e.g., use torch tensor instead
     of numpy array).
Make sure any new tensor or variable you introduce is on the same device as the input tensors.
```

Figure 2.2: Eureka Initial prompt.



```diff
  def compute_reward(object_rot, goal_rot, object_angvel, object_pos, fingertip_pos):
      # Rotation reward
      rot_diff = torch.abs(torch.sum(object_rot * goal_rot, dim=1) - 1) / 2
-     rotation_reward_temp = 20.0
+     rotation_reward_temp = 30.0
      rotation_reward = torch.exp(-rotation_reward_temp * rot_diff)

      # Distance reward
+     min_distance_temp = 10.0
      min_distance = torch.min(torch.norm(fingertip_pos - object_pos[:, None], dim=2), dim=1).values
-     distance_reward = min_distance
+     uncapped_distance_reward = torch.exp(-min_distance_temp * min_distance)
+     distance_reward = torch.clamp(uncapped_distance_reward, 0.0, 1.0)

-     total_reward = rotation_reward + distance_reward
+     # Angular velocity penalty
+     angvel_norm = torch.norm(object_angvel, dim=1)
+     angvel_threshold = 0.5
+     angvel_penalty_temp = 5.0
+     angular_velocity_penalty = torch.where(angvel_norm > angvel_threshold,
+         torch.exp(-angvel_penalty_temp * (angvel_norm - angvel_threshold)), torch.zeros_like(angvel_norm))
+
+     total_reward = 0.5 * rotation_reward + 0.3 * distance_reward - 0.2 * angular_velocity_penalty

      reward_components = {
          "rotation_reward": rotation_reward,
          "distance_reward": distance_reward,
+         "angular_velocity_penalty": angular_velocity_penalty,
      }

      return total_reward, reward_components
```

Figure 2.3: EUREKA is capable of zero-shot generating executable reward functions and can iteratively refine them through various types of flexible, free form modifications including (1) adjusting hyperparameters of existing components, (2) altering their functional structure and (3) adding entirely new reward components.



```
### Action Space
The action is a `ndarray` with shape `(1,)` which
can take values `{0, 1}` indicating the direction
of the fixed force the cart is pushed with.
    | Num | Action               |
    |---- |------------------- |
    | 0   | Push cart to the left|
    | 1   | Push cart to the right|

### Observation Space
    The observation is a `ndarray` with shape `(4,)`
with the values corresponding to the following
positions and velocities:
    | Num | Observation          | Min           | Max
    |-----|--------------------- |---------------|--------
    | 0   | Cart Position         | -4.8          | 4.8
    | 1   | Cart Velocity         | -Inf          | Inf
    | 2   | Pole Angle            | -24°          | 24°
    | 3   | Pole Angular Velocity | -Inf          | Inf
```

We trained a RL policy using the provided reward function code and tracked the values of the individual components in the reward function as well as global policy metrics such as success rates and episode lengths after every 300 epochs and the maximum, mean, minimum values encountered: upright_reward: ['0.03', '0.31', '0.30', '0.32', '0.34', '0.34', '0.34', '0.34', '0.34', ' 0.32'], Max: 0.36, Mean: 0.32, Min: 0.03 Please carefully analyze the policy feedback and provide a new, improved reward function (...)

The agent's task is to keep a pole balanced upright on a moving cart by applying left or right forces to the cart. The goal is to prevent the pole from tipping over or the cart from moving too far off-center for as long as possible.

```
# Here is a function that compute
a reward for the Cartpole
Environment generated by the LLM:
def compute_reward(observations)
# Reward for keeping the pole
upright
upright_reward = ...
(...)
return upright_reward
```

Figure 2.4: Eureka Concept Scheme.

Figure 2.4 provides a good summary of the Eureka concept. It illustrates how the Eureka process works using the **CartPole** environment. In this simple environment, the goal is to keep a pole upright by moving left or right. For more details about it, visit this link.

This diagram represents the flow of an Eureka experiment. The LLM box positioned in the center-bottom area, identifiable by the black-corner marker symbolizes the interaction with the LLM. The process begins by providing the environment and task description to the LLM, so it understands where and how the reward function will be applied. Based on this information, the LLM generates an initial reward function (highlighted in red in the diagram).

This function is then used in RL training (blue section), where the agent learns to complete the task by interacting with the environment. During training, various performance metrics related to the reward are tracked. These metrics are essential for the reward reflection process (part in purple), which helps the LLM refine its response based on the previous iteration. The updated reward function should improve learning efficiency and task performance and solve potential previous errors.

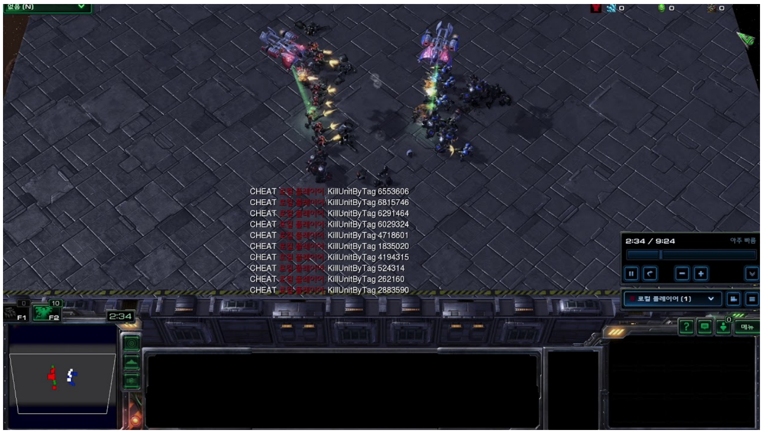
This cycle is repeated for as many iterations as desired. However, training RL models is computationally expensive and requires tuning many parameters, so, in practice, we limit the number of iterations to keep the process manageable.

2.5 PyTorch

PyTorch[20] is one of the technologies used by the Eureka framework. PyTorch is an open source machine learning library that is used to build and train neural networks. It allows developers and researchers to easily create deep learning models by providing flexible tools to handle data, define models, and run them efficiently on both CPUs and GPUs. PyTorch is primarily available in Python. It is widely used because of its simplicity, dynamic computation graphs (which allow for easy debugging), and strong support for tasks like image recognition, natural language processing and RL.

2.6 SMAC

The StarCraft Multi-Agent Challenge [21] (SMAC) is a benchmark framework designed to test and evaluate the performance of MARL algorithms in the real-time strategy game StarCraft II. It focuses on decentralized control, where each unit in a team acts independently based on local observations to achieve a shared goal, such as defeating an opponent team. The SMAC environment presents complex challenges in coordination, strategy, and decision making, making it a popular testbed for advancing MARL research. Figure 2.5 presents an example scenario within a SMAC environment in StarCraft II.



```
CHEAT 로컬 플레이어 KillUnitByTag 6553606
CHEAT 로컬 플레이어 KillUnitByTag 6815746
CHEAT 로컬 플레이어 KillUnitByTag 6291464
CHEAT 로컬 플레이어 KillUnitByTag 6029324
CHEAT 로컬 플레이어 KillUnitByTag 4718601
CHEAT 로컬 플레이어 KillUnitByTag 1835020
CHEAT 로컬 플레이어 KillUnitByTag 4194315
CHEAT 로컬 플레이어 KillUnitByTag 524314
CHEAT 로컬 플레이어 KillUnitByTag 262160
CHEAT 로컬 플레이어 KillUnitByTag 2883590
```

Figure 2.5: SMAC Environment Visualization Example.

2.6.1 PyMARL

PyMARL is a toolkit for training and testing different teamwork algorithms with SMAC. SMAC is all about teaching multiple units in StarCraft II to work together without needing a central controller. PyMARL connects easily with SMAC, letting researchers quickly try out new strategies and see how well different approaches help these units work as a team in a game. You can access the PyMARL website: https://github.com/oxwhirl/pymarl.

2.6.2 EPyMARL

EPyMARL is an extended version of PyMARL, aimed at improving modularity, scalability, and functionality. It introduces support for a broader set of environments, algorithms, and configurations, addressing some of the limitations of the original framework. It is mainly used in this work and has been in others [22]. You can learn more about it here.

2.6.3 Mava

Mava [23] is a research framework for training MARL agents, similar to EPyMARL and PyMARL. Developed by Instadeep, Mava is built on top of Acme (by DeepMind) and is designed to support scalable and reproducible MARL experiments. In this work, Mava has been used to run and manage experimental setups for training and evaluating MARL agents.

2.6.4 SMACLite

SMACLite is a lightweight variant of SMAC [24], designed to simplify the testing and benchmarking of MARL algorithms. While SMAC is built on top of the full StarCraft II engine and involves complex cooperative tasks, SMACLite significantly reduces the computational and resource overhead by abstracting the StarCraft II environment into a simpler, grid-based framework. Both Mava and EPyMARL allow to train agents in this environment.

Default Reward

SMACLite comes with a default reward function, designed by humans to train agents within this environment. This built-in reward will serve as the baseline for comparing the performance of the LLM-based reward function.

The default reward function operates as follows: After each time step, all agents receive a shared reward based on the amount of health and shield points they remove from enemy units. They also get a small bonus of 10 points for every enemy they eliminate and a larger bonus of 200 points for winning the scenario. To keep the scale consistent, the total reward is normalized by dividing it by the maximum possible damage and bonuses, then multiplied by 20. This keeps most rewards between 0 and 20. However, due to health and shield regeneration during episodes, the final cumulative reward can sometimes go beyond 20.

The default reward in SMAClite encourages agents to learn aggressive and efficient combat behavior. Because they receive rewards for reducing enemy health and shield, the agents are driven to actively engage and consistently inflict damage rather than remain passive. The additional bonus for eliminating enemies teaches them to prioritize finishing off weakened opponents, promoting focused attacks instead of spreading damage across multiple targets. Moreover, the large bonus granted for winning the scenario motivates agents to act strategically and cooperatively to ensure overall success, not just individual combat performance. As a result, agents learn to balance damage output, target prioritization, and survival in order to consistently win combats.

Observation Space

At every time step, each agent receives an observation of what it perceives in the environment. SMACLite uses a fixed sight range, if another unit is too far away or already dead, the agent gets no information about it (this data is represented as zeros).

The observation vector is divided into four parts:

- **Movement information**: Indicates whether the agent can move in each of the four cardinal directions (up, down, left, right).
- **Enemy information**: For each enemy unit, the agent receives information on whether the unit is alive, if it can be attacked, its distance and relative position (X and Y coordinates), its health, shields (if applicable), and a one-hot encoded vector representing its unit type (if applicable).
- **Ally information**: Similar to the enemy section, the agent receives data for each teammate, including whether they are visible (a value of 1), their distance, relative position, health, shields (if applicable), and unit type (if applicable).
- **Self information**: This includes the agent's own health, shields (if applicable), and its unit type (if applicable).

All values in the observation vector are normalized to fall within the range $[0, 1]$, similar to the state features. The shield is applicable if at least one unit has shield, and unit type is applicable if the scenario contains different type of units (zergs and marines for example).

Example:
Let's take a simple scenario where we have 4 marines (allies) against 3 marines (enemies), none of the units have shield and there is only marines. This is a quite easy scenario. The full observation for this unit would be like this:

```
[1, 0, 1, 1,                  # Movement information
 1, 0.5, 0.1, -0.3, 0.8,      # Enemy 1
 1, 0.7, -0.2, 0.1, 0.6,      # Enemy 2
 0, 0, 0, 0, 0                # Enemy 3 (dead or out of range)
 1, 0.3, 0.5, -0.2, 0.9,      # Ally 1
 1, 0.2, -0.1, 0.4, 1.0,      # Ally 2
 0, 0, 0, 0, 0                # Ally 3 (dead or out of range)
 0.75]                        # Own Health
```

The information we can extract from this example is:

- The agent can move **up**, **left**, and **right**, but not **down**.
- The first enemy is attackable. He is at a normalized distance of 0.5 from the agent, with a position of $(x = 0.1, y = -0.3)$ and a health value of 0.8.
- The second enemy is also attackable. He is at a normalized distance of 0.7 from the agent, with a position of $(x = -0.2, y = 0.1)$ and a health value of 0.6.
- The third enemy is either dead or out of range.

- The first ally is alive and within range. He is at a normalized distance of 0.3, located at ($x = 0.5$, $y = -0.2$), with a health value of 0.9.
- The second ally is alive and within range. He is at a normalized distance of 0.2, located at ($x = -0.1$, $y = 0.4$), with a health value of 1.0.
- The third ally is either dead or out of range.
- The health of the current agent is 0.75.

This example corresponds to the observation of a single agent. In practice, the full observation includes the states of all allied agents. In our case, the observation consists of a list of 4 elements, as there are 4 allied agents in total.

Action Space

Each agent in SMACLite has a set of possible actions it can take, but not all actions are available at every time step. The environment provides a method to check which actions are currently valid for each agent. If an agent tries to perform an invalid action, the environment throws an error and stops the simulation.

The available actions include:

- **no-op**: Does nothing and is only available for dead units.
- **stop**: Tells the unit to stop and stay in place.
- **moveN, moveE, moveS, moveW**: Commands the unit to move north, east, south, or west.
- **target1, target2, ...**: Tells the agent to target a specific unit based on its team-specific ID. For attacking units, this means attacking an enemy. For healing units, it means healing an ally.

Agents can only target units that are within a fixed range defined by SMACLite. If a target is outside this range, the targeting action becomes unavailable.

Example:
If we keep the same scenario as before (4 marines VS 3 marines), the action space looks like this:

```
0 ==> NO-OP (Only for dead units)
1 ==> STOP
2 ==> MOVE_NORTH
3 ==> MOVE_SOUTH
4 ==> MOVE_EAST
5 ==> MOVE_WEST
6 ==> ATTACK enemy 0
7 ==> ATTACK enemy 1
8 ==> ATTACK enemy 2
```

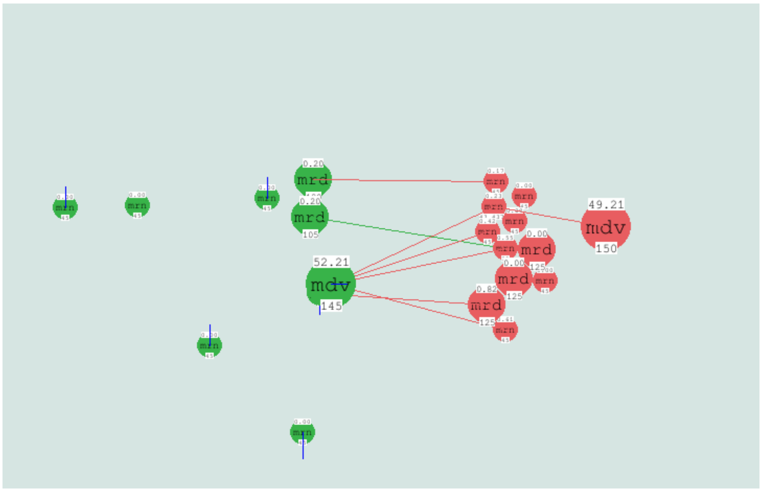


Figure 2.6: SMAClite Environment Visualization.

The image above shows a visualization of the SMAClite environment in a more complex scenario than the one described earlier (4vs3). The green circles represent allied units, and the red ones represent opponents. Each unit has an acronym indicating its type: mrn for marine, mrd for marauder, and mdv for medivac. The size of each circle is proportional to the unit's size in StarCraft II.

The number below each unit shows its health + shield, while the number above represents the unit's cooldown. The Cooldown is the time a unit must wait after attacking before it can attack again it reflects a realistic attack speed.

2.7 Prompt Engineering

In this work, prompt engineering is very important because the performance of LLMs heavily depends on how the input is formulated. Well-designed prompts can guide the model to produce more accurate, relevant, and coherent responses. On the other hand, wrong prompts can mislead the LLM answers. In RL or decision-making settings, effective prompting helps the model better understand the environment, task objectives, or desired behavior, ultimately improving learning outcomes and agent performance. For this reason, we will see in this section what prompt engineering consist of and how it is applied to Eureka prompts. The source of this knowledge comes from Prompt Engineer in Medical Education [25].

What is Prompt Engineering?

Prompt engineering is the process of designing inputs (called prompts) that help artificial intelligence (AI) models, such as chatbots or virtual assistants, provide the best possible responses.

Types of Prompts

Prompts can be designed in different ways depending on how much guidance is provided:

1. **Zero-shot Prompts**: These prompts provide no examples, relying entirely on the AI's pre-existing knowledge. For instance, asking, "Describe the importance of photosynthesis" without offering additional guidance.
2. **Few-shot Prompts**: These prompts include examples to guide the AI's response. For example, "Here is an example of a simple explanation of a biological process: [example]. Now, describe the importance of photosynthesis."

Prompting Levels

Prompts can be designed with varying levels of detail to improve the quality of responses:

1. **Level 1**: Simple questions, such as "What is machine learning?"
2. **Level 2**: Add more context or role-playing, such as, "You are a professor. Explain machine learning to a group of high school students."
3. **Level 3**: Include examples or specific styles, for example, "Here is a lecture slide I found helpful. Explain machine learning in a similar way."
4. **Level 4**: Encourage step-by-step problem solving, like asking the AI to "Explain the steps involved in training a machine learning model step by step."

Structured Prompts

The best results come when prompts include:

- A description of who is asking the question or their context (e.g., "I am a software engineer exploring reinforcement learning concepts").
- A clear task description (e.g., "Explain the purpose of reward functions in reinforcement learning.").
- A role for the AI (e.g., "Act as an experienced AI researcher with expertise in reinforcement learning.").
- Specific instructions for the response (e.g., "Provide a concise explanation in no more than 150 words, using simple language suitable for beginners").

Iterative Prompts

Sometimes, the best prompt is created in steps. You can give the AI feedback on its responses to refine your original prompt. This back-and-forth process is called iterative prompting.

What are Bad Prompts?

Not all prompts work well. Here are examples of common problems:

- **Vague Prompts**: Questions like "What is the universe?" or "What is the meaning of life?" are too broad and can lead to unclear or unhelpful responses.
- **Overly Complex Prompts**: Combining multiple queries, such as "Explain climate change and list all solutions for renewable energy," may overwhelm the AI and reduce response quality.

- **Logical or Math Puzzles**: Tasks like solving detailed math problems can result in errors unless guidance is provided, such as "Break down the solution step by step."
- **Fake References**: AI may invent citations or references, so it is important to validate any sources included in its responses.

2.7.1 Application to Eureka

In Eureka we can clearly see that some prompt Engineering techniques are used to improve the response of the LLM. Let's take the example of the Initial prompt from the figure 2.3:

- The prompt clearly defines the goal of designing a reward function to help a RL agent learn a task effectively, providing context that guides implementation.
- Adherence to `@torch.jit.script` ensures compatibility and precision by enforcing computational constraints, avoiding vague or ambiguous instructions.
- The clear function signature promotes iterative refinement, enabling gradual improvement in achieving an optimal reward function for RL tasks.

Prompt 3: Code formatting tip

```
The output of the reward function should consist of two items:
    (1) the total reward,
    (2) a dictionary of each individual reward component.
The code output should be formatted as a python code string: "```python ... ```".

Some helpful tips for writing the reward function code:
    (1) You may find it helpful to normalize the reward to a fixed range by applying
    transformations like torch.exp to the overall reward or its components
    (2) If you choose to transform a reward component, then you must also introduce a
    temperature parameter inside the transformation function; this parameter must be a named
    variable in the reward function and it must not be an input variable. Each transformed
    reward component should have its own temperature variable
    (3) Make sure the type of each input variable is correctly specified; a float input
    variable should not be specified as torch.Tensor
    (4) Most importantly, the reward code's input variables must contain only attributes of
    the provided environment class definition (namely, variables that have prefix self.).
    Under no circumstance can you introduce new input variables.
```

Figure 2.7: Eureka Code Formatting tip prompt.

This prompt effectively demonstrates the principles of clear and structured design. It specifies a well-defined output format, emphasizes normalization for consistent reward scaling. By limiting inputs to environment-specific variables and ensuring correct type specifications, it fosters compatibility and minimizes errors.

Prompt 2: Reward reflection and feedback

```
We trained a RL policy using the provided reward function code and tracked the values of the
     individual components in the reward function as well as global policy metrics such as
     success rates and episode lengths after every {epoch_freq} epochs and the maximum, mean,
     minimum values encountered:
<REWARD REFLECTION HERE>

Please carefully analyze the policy feedback and provide a new, improved reward function that
     can better solve the task. Some helpful tips for analyzing the policy feedback:
    (1) If the success rates are always near zero, then you must rewrite the entire reward
     function
    (2) If the values for a certain reward component are near identical throughout, then this
     means RL is not able to optimize this component as it is written. You may consider
        (a) Changing its scale or the value of its temperature parameter
        (b) Re-writing the reward component
        (c) Discarding the reward component
    (3) If some reward components' magnitude is significantly larger, then you must re-scale
     its value to a proper range
Please analyze each existing reward component in the suggested manner above first, and then
     write the reward function code.
```

Figure 2.8: Eureka Reward reflection and feedback prompt.

We can see that this prompt exploits the concept of iterative prompting. It provides the LLM with feedback about how well its rewards perform during training, asks it to analyze the tracked metrics, and adapt its code accordingly. By doing so, we obtain a refined reward function.

2.8 Double Bias

The approach we use in this work introduce a double bias:

The First [26] is when we write prompt to the LLM we introduce Bias into the answer of the LLM. The language and structure of the prompt we provide to the LLM will influence the kind of answers it generates. For example, if we use a LLM to generate reward function (just like we did in this work), if our prompt suggests that the reward function should prioritize efficiency, fairness, or some other specific value, the LLM may "bias" its output toward that idea, even if that wasn't the most appropriate reward structure for the task.

The Second [27] comes from the Reward Function to the RL agent. Once we have a reward function created (either by the LLM or manually), this function directly influences the behavior of the RL agent. The reward function bias can then lead to the agent optimizing for the wrong goal, this the Reward Hacking concept explained earlier.

One thing in the eureka process that helps reduce these biases is using iterative prompts, where we ask the model to refine its answers. This approach can help mitigate bias because it allows us to adjust and improve the reward function over time, ensuring it better aligns with the true goal and reducing the chances of unintended results. However it is important to keep in mind that those bias are present.

3 Eureka Implementation

In this chapter, we describe how the Eureka codebase, originally developed by ***Eliot Crancée***, has been adapted to facilitate our goal of testing LLM-generated reward functions within the SMAClite environment using the Eureka process. These modifications are driven by the goal of achieving our objectives and, while not central to the fundamental concepts of the thesis, they provide insight into the low-level processes behind each experiment.

3.1 Prompts

One of the first steps is to modify the prompts given to the LLM. These prompts must be carefully crafted based on the concepts of Prompt Engineering discussed earlier, in order to enhance the efficiency of the LLM when performing the task. The prompts are based on the one from Eureka.

Most of the prompts can be kept the same for the task of generating a reward function since they are quite generic. For example, error feedback or initial system prompts can remain unchanged. On the other hand, we need prompts to describe the SMAClite environment with the new observation and action space to the LLM and also a new task description which changed according to the specific task.
The Environment Description needs to explain:

1. How is structured the observations manipulated in the SMAClite Environment
2. What are the possible actions an agent can take in the Environment
3. How an episode might ends

All this information is contained in the *env_desc.txt* file and given to the LLM so that it can understand the task we ask it to do. The point 1 and 2 consist of explaining the observation space and action space described earlier. The prompts for the experiments can be found in the appendices

An episode terminates when one of the following conditions is met:

- All Enemies are Defeated.
- All Allied Units are Defeated
- Time Limit exceeds (max step reached).

3.2 Mava and EPyMARL Modifications

One of the key modifications required for the SMAClite Environment is the ability to use a custom reward function instead of the default one. To achieve this, the SMAClite Wrapper classes have been modified accordingly both in Mava and ePyMARL. Additionally, it was necessary to adapt those classes to track reward components, allowing the reward reflection of Eureka to work properly. Furthermore, for ePyMARL, changes were made to allow the recording of specific episodes when needed.

For ePyMARL, the RL training process remains largely the same, but with some modifications. An early stopping mechanism has been implemented to reduce training time and computational cost. Since RL training is time intensive and the Eureka framework requires extensive training, continuing a learning process that is unlikely to yield good results is inefficient. To address this, the early stopping mechanism was introduced to save both time and computational resources. The idea is simple: If during training, the model fails to achieve at least one successful episode within a given number of timesteps (learning steps), the training is stopped. This ensures that models with insufficient reward signals do not continue learning unnecessarily, preventing wasted computation on unpromising policies.

For Mava, the only changes applied were within the wrapper, with no significant modifications made to the rest of the codebase.

3.3 Important files Explanation

3.3.1 eureka.py

The provided script forms the heart of the program where the Eureka experiments are launched and executed. It is responsible for setting up the experiment environment, initializing the necessary configurations, running the iterations and collecting results.

Overview of the Core Workflow

The core of the Eureka experiment is implemented in the `main` function. The sequence of operations can be described as follows:

- **Load Configuration:** The script starts by loading the YAML configuration file, `experiment_config.yaml`, which contains the environment details, experience settings, and model configurations.
- **Initialize Experiment:** The `init_experiment` function is called to initialize the directories and files required for the experiment. This function sets up directories for prompts, environment configurations, descriptions, and the experience. If these directories do not exist, an exception is raised.
- **Load Prompts and Model:** Once the experiment is initialized, the prompts and models are loaded, as specified in the configuration.
- **Run Eureka Experiment:** The script then proceeds to run the Eureka experiment through the `run_eureka_experiment` function. This function handles the iterative training process, model evaluation, and rewards collection.
- **Plot Results:** After the experiment completes, the `plot_eureka_training` function generates visualizations, such as success rate evolution and best model performance, to summarize the experiment's results.

Iteration and Model Selection

During the experiment, multiple iterations of the Eureka framework are performed, with each iteration generating several samples. At the end of each iteration, the model with the highest evaluation score (based on success rate) is selected as the best-performing model. The experiment tracks the performance across iterations, and the results are visualized plots as explained before.

Conclusion

In summary, this script is at the heart of the Eureka experiment, orchestrating the initialization, execution and visualization of models.

3.3.2 LLM.PY

This Python file is designed to load different LLM using specific configurations and API keys. It defines functions to interact with multiple language models like Groq and GPT-models.

3.3.3 EPYMARL_MAPPO.PY

This Python script is designed to train and evaluate the MAPPO model using the EpyMarl framework in the Smaclite environment. It handles the process of training the model, running simulations, and evaluating its performance by calculating reward components and various statistical metrics such as mean reward, success rate, and others. Additionally, the script supports the recording and processing of videos from the simulation, including a slow-motion feature for further analysis of the agent's actions. The operations in the script are structured to facilitate multiple training iterations, evaluate the agent's performance across these iterations and visualize both numerical metrics and video feedback for model analysis and improvement.

3.3.4 MAVA_IPPO.PY

This script trains an IPPO model using Mava in the SMACLite environment.the train_and _evaluate function handles the training by calling Mava's *ff_ippo.py* script with the needed parameters. After training, it reads performance data (like win rate and average return) from a JSON file and creates plots to visualize the results. It also analyzes reward data by calculating statistics such as the average, standard deviation, minimum, maximum, and all recorded values. These results are saved in a JSON file for future use with eureka.

3.3.5 EXPERIMENT_CONFIG.YAML

This configuration file is used for setting up the parameters of the Eureka experiment in the Smaclite environment. Below is a breakdown of each hyperparameter:

- `env_name`: Specifies the name of the environment used in the experiment. In our case, it is set to "Smaclite" and stay fixed. (this is for further testing with different environment)
- `map_name`: Defines the map to use in the experiment, it can be seen as the SMAClite scenario.
- `experience_name`: Defines the name of the experiment.
- `prompt_name`: Indicates the name of the prompt configuration used for the LLM. There is different possible versions of prompts available that might change according to the environment used.
- `env_description_version`: Specifies the version of the environment description being used. Same as the prompts, the environment description changes according to the `map_name` (scenario).
- `llm`: Specifies the type of large language model used in the experiment. This parameter determines which model the system will interact with.

- `eureka_iter`: Represents the number of iterations for the Eureka experiment, which aims to fine-tune the behavior of the system using reward reflection with feedback.
- `eureka_samples`: Specifies the number of samples to collect in each Eureka iteration.
- `rl_timesteps` (EPyMARL): Specifies the number of timesteps for training the RL model.
- `last_model` (EPyMARL): Specifies the path to the last trained model. If this parameter is left empty, a new model will be trained from scratch.
- `last_model_timesteps` (EPyMARL): Defines the number of timesteps to resume training from, in case a pre-trained model is used. A value of 0 indicates that the model will be loaded from the latest available timesteps.
- `early_stopping` (EPyMARL): whether or not to use the default reward of SMAClite.
- `early_stopping_patience` (EPyMARL): The maximum number of steps allowed without any wins occurring. If this number is reached without any wins, the training stops. This is only useful if `early_stopping` is true.
- `use_default_reward`: whether or not to use the default reward of smaclite.
- `time_limit`: The maximum number of steps allowed for an episode of Smaclite.
- `time_interval_tracking` (Mava): time interval we track the reward metrics (in seconds), meaning that every x seconds we will save the reward metrics for one episode during training.
- `num_updates` (Mava): number of updates for the training with mava.

4 Experimentation

This chapter presents the experimental setup and methodology used to evaluate the effectiveness of LLM reward function generation in cooperative MARL environments.

We begin by describing the overall structure of an Eureka experiment and the integration of LLMs with RL agents. The models tested, prompt configurations, and reward refinement process are detailed to give insight into the system pipeline. This is followed by a description of the three SMAClite scenarios used, each representing a different level of difficulty and complexity. We then explain our strategy for ensuring experiment stability. Special attention is given to how specific micro-strategies (e.g., Focus Fire and Dancing) are injected into prompts to guide the reward shaping process. The final sections detail the training and evaluation procedures under both EPyMARL and Mava frameworks, including iterative refinement cycles and metrics collected. This chapter ultimately provides the foundation for the result analysis in the next chapter by laying out how data was produced and collected.

4.1 Conduct Of an Experiment

Eureka experiments integrate RL and LLMs to enhance AI decision-making. The core idea is to iteratively refine reward functions using LLM-generated suggestions and RL feedback. This section details how a experiment is structured and conducted.

Experiment Setup

A Eureka experiment is configured using a YAML file that defines parameters such as explained in the experiment_config.yml section. Before execution, necessary directories are created, and configuration files are stored.

Loading Prompts

Prompts from predefined text files are loaded, including system instructions, reward function descriptions, and feedback templates.

LLM Used

The Models tested in this work are the following:

- GPT-4o (gpt-4o-2024-08-06)
- GPT-4o (latest version: chatgpt-4o-latest)
- O3-mini (o3-mini-2025-01-31)
- GPT-4.1 (gpt-4.1-2025-04-14)
- O4-mini (o4-mini-2025-04-16)

The selection of models for this work is guided by the goal of comparing performance across a representative range of recent and efficient LLMs. We include `GPT-4o` (gpt-4o-2024-08-06) and its latest variant `chatgpt-4o-latest` due to their optimized architecture for both speed and quality, offering state-of-the-art performance while being highly responsive for real-time, and complex tasks. The inclusion of `O3-mini` (o3-mini-2025-01-31) and `O4-mini` (o4-mini-2025-04-16) allows for the evaluation of more lightweight models, which are designed to balance performance with lower computational cost. These models are particularly valuable when considering deployment in resource-constrained environments. Finally, `GPT-4.1` (gpt-4.1-2025-04-14) is used as a strong and stable baseline representing the evolution of OpenAI's full-scale GPT-4 architecture. This diverse model set enables us to assess how different generations and sizes of LLMs handle complex tasks such as reward design and prompt-guided learning in MARL scenarios.

In this experiment, the temperature parameter is used to control the randomness and creativity of the LLM responses. A higher temperature encourages more diverse and exploratory outputs, while a lower temperature results in more focused and deterministic answers. In our setup, a temperature of 0.85 is applied to non-mini models to allow for a balance between creativity and coherence, encouraging the generation of varied reward function designs. For the mini models, the temperature parameter cannot be customized, as these models use fixed default settings to ensure lightweight and efficient performance.

4.2 Experiments Description

4.2.1 SMAClite Scenarios Tested

In this work, we evaluated our approach on three different scenarios using the SMAClite environment, each with increasing levels of difficulty and complexity. All scenarios are limited to a single unit type: Marines and no unit has shields. This simplification reduces the complexity of the observation space and focuses the learning task on spatial positioning and action coordination rather than unit diversity or shield management.

The first scenario is a **4 Marines vs 3 Marines** setup. This is considered an easy scenario due to the numerical advantage of the allied agents. It serves as a baseline to validate whether the models are capable of learning with the LLMs rewards when they hold a clear advantage.

The second scenario is a **3 vs 3 Marines** setting. Here, the battle is numerically balanced, removing the advantage and presenting a slightly more challenging coordination task.

The third and most challenging scenario is **10 Marines vs 11 Marines**. In this case, the agents are at a numerical disadvantage, and the environment features has a larger number of units, which increases the complexity of both the observation and action spaces.

4.2.2 Experiment Repetition and Stability Considerations

To address the inherent instability of RL and the variability introduced by LLM generations, most experiments in this work were performed at least three times. This repetition aims to ensure a minimum level of consistency and reliability in the observed results.

RL algorithms are known to be very unstable. As a result, performance can vary significantly across training runs, even when all parameters remain the same. Similarly, LLMs can produce different outputs across queries, contributing further to the variability of outcomes.

Combining these two components introduces compounded instability into the learning pipeline. The generated reward functions may lead to different learning dynamics in each run, making reproducibility and convergence more challenging. By running each configuration multiple times, we aim to average out outliers, capture general trends, and better understand the robustness of the approach under varying conditions.

4.2.3 Suggested Micro Strategies in Prompt

In the experiments, different prompts were provided to the LLM to evaluate whether guiding it with specific micro-strategies could lead to the generation of more effective reward functions or not. The remaining part of the prompt remains identical across experiments within the same scenario, as each scenario requires a distinct environment description due to its specific characteristics (number of units).

Focus Fire prompt given to the LLM:

> Use this Strategy: To improve combat efficiency in SMACLite, the agent should prioritize a focus fire strategy, where multiple units coordinate their attacks on a single enemy at a time. This approach maximizes damage output, eliminates threats faster, and reduces incoming damage by quickly lowering the number of active enemies. The reward function should encourage this behavior by providing higher rewards when units collectively target and eliminate an enemy rather than distributing attacks across multiple enemies.

Dancing prompt given to the LLM:

> Use this Strategy: Retreating individual units that are taking fire to allow them to live longer will increase the time it takes for your opponent to kill your army, allowing you more time to kill his army. After they are no longer under attack they can be ordered to join the fight again. This technique is called Dancing.

One important note is that the LLM is used to generate reward functions based on the prompt it receives, while the strategies like *Focus Fire* or *Dancing* describe specific tactical behaviors expected from the agents. The LLM does not execute these strategies directly, it only guides the training by shaping the reward function. Whether or not the agents actually learn these strategies depends on how well the reward function encourages them to do so.

Experiment Description Summary

The LLM-generated rewards have been tested using different hyperparameters related to the LLM itself, the prompt provided to the LLM, and the scenario used in SMACLite. Most experiments were repeated at least three times to make our results more consistent. We did this because RL training involves some randomness and can be unstable.

Here are the different values for each hyperparameter:

- **LLM**: chatgpt-4o-latest, gpt-4o-2024-08-06, o3-mini-2025-01-31, o4-mini-2025-04-16 and gpt-4.1-2025-04-14
- **Strategies mentioned in the prompt**: Focus Fire, Dancing, or no strategy provided
- **Scenario**: 4 marines vs 3 marines, 3 marines vs 3 marines, and 10 marines vs 11 marines

Initial Training and Evaluation

The RL algorithm (MAPPO in EPyMARL and IPPO in Mava) is initialized and trained with a reward function given by the LLM.

Iterative Improvement with LLMs

Each iteration begins with the LLM proposing a new reward function sample. This sample is then tested within the RL environment. Following this, performance metrics are collected and compared across different samples. The best-performing reward function among those generated is selected to guide the next reward reflection phase.

Model Training and Evaluation with EPyMARL

The training and evaluation process using EPyMARL begins by training a MAPPO model with the custom reward function provided by the LLM. Once training is complete, the model is evaluated to gather detailed values of the reward components, which are used in the reward reflection process. This evaluation phase includes computing success rates and reward metrics across multiple episodes typically ten in practice. Additionally, videos are recorded to capture the agent's behavior, allowing visual assessment of the learned policy's effectiveness and its ability to solve the task.

Model Training and Evaluation with Mava

Using the Mava framework, the IPPO model is trained with the reward function generated by the LLM. During this training, reward components are tracked at regular intervals defined by the time_interval_tracking parameter in the experiments_config.yaml file. After the training phase, learning performance is evaluated by plotting key metrics such as the average reward and win rate, which help assess the quality of the learning process.

5 Results & Analysis

This chapter presents and analyzes the experimental results obtained throughout this master's thesis, using both the EPyMARL and Mava frameworks. The analysis begins with an evaluation of the default reward function provided in the SMAClite environment, which serves as a baseline for comparison with the reward functions generated by LLMs. This baseline allows us to assess whether the LLM-generated rewards lead to improved learning outcomes or not.

Subsequently, we evaluate the performance of RL models trained using LLM designed reward functions across multiple SMAClite scenarios. We compare the impact of different LLM configurations and prompt strategies on learning behavior, using metrics such as win rates and reward evolution. This analysis highlights strengths, limitations, and trends observed throughout the experiments. By the end of the chapter, we will gain a comprehensive understanding of how reward design especially when assisted by LLMs affects training performance in SMAC environments.

5.1 EPyMARL Results

5.1.1 Default Reward

This section presents the results of training agents in a SMACLite environment using EPy-MARL along with the default reward function from the basic SMACLite implementation. A description of this reward function can be found here. The results shown below correspond to the training of three agents with the Default Reward and the MAPPO algorithm. The maximum number of steps is set to 500; if this limit is exceeded, the episode ends. The training has been done during a total of 1 000 000 timesteps (total of steps done across all episodes).

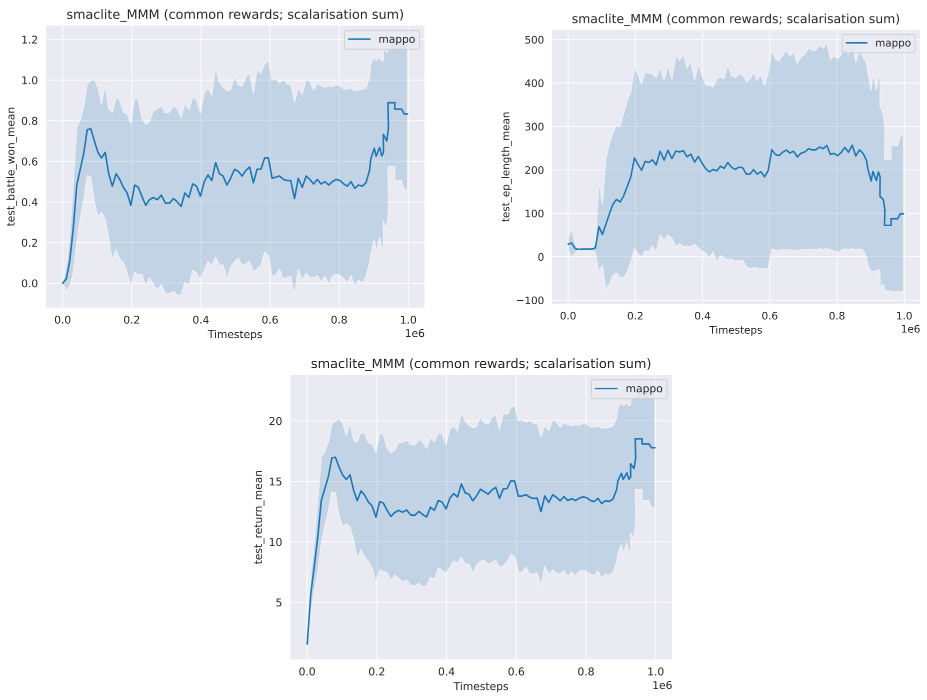


Figure 5.1: Default Reward in a 3 marines VS 3 marines scenario with mappo algorithm

The first graph shows the evolution of the *win rate* across timesteps. The metric being tracked is *test_battle_won_mean*, which is measured every 10,000 timesteps over 10 episodes (outside of training) to assess how well the agent has learned to win battles. We can observe a high variance, which is concerning given that this is the default reward. However, the *win rate* appears to improve with more timesteps.

The second graph displays the mean episode length during testing. Again, we see high variance. This metric can be interpreted in a few ways. If the episode length is too short, it likely indicates that the allied units died quickly. On the other hand, if the length is too long, it suggests that some allies survived but took too long to defeat the enemies, which is also undesirable. Ideally, the episode length should be around 200 to 300 timesteps.

The final graph shows the average reward given to the agents. As expected, this is proportional to the win rate, since a higher reward typically means the agent is getting closer to winning. Like the previous graphs, this one also displays significant variance.

That was for a 3 vs 3 scenario we also to test in a even simpler scenario of 4 vs 3 which should be more easier since the allies have the advantage of the number. In this case, let's just analyze the win rate and mean number of episode length. Once again the results are not stable at all. Unfortunately, We can't see a clean learning curve.

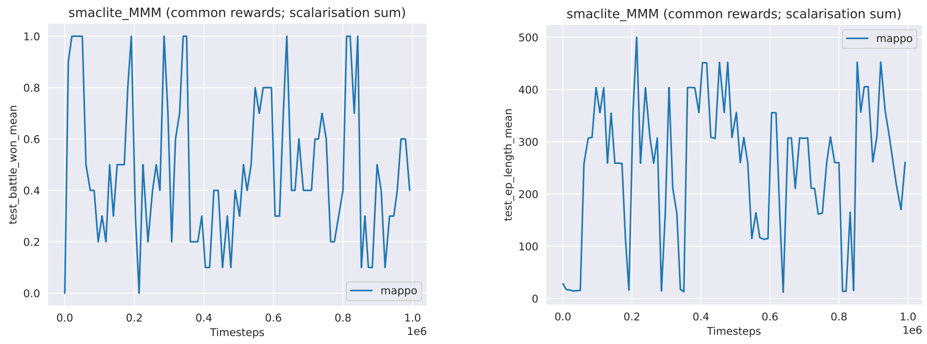


Figure 5.2: Default Reward in a 4 marines VS 3 marines scenario with mappo algorithm

Test with QMIX and IPPO

It’s possible that the problem comes from the training algorithm rather than the reward function. To test this idea, we ran experiments using the QMIX algorithm. We repeated the same experiment in a 4 marines versus 3 marines scenario three times and tracked only the win rate to see if the results were more stable.

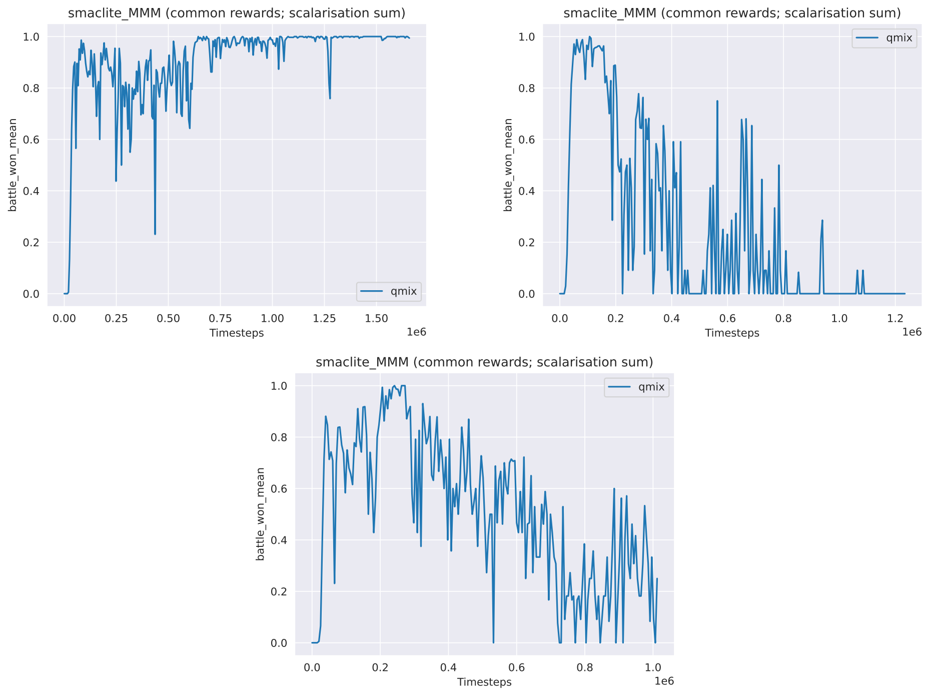
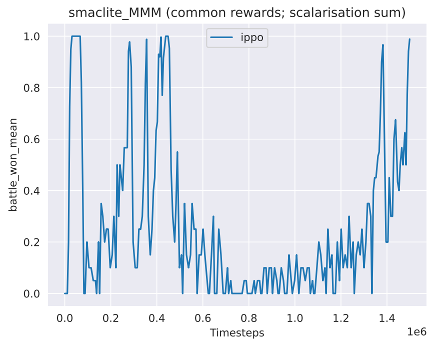


Figure 5.3: Default Reward in a 4 marines VS 3 marines scenario with QMIX algorithm

Although the first experiment showed promising results, the other two did not produce consistent outcomes. In fact, in the last two runs, the win rate dropped as the number of time steps increased. All three experiments showed high variance in win rate, suggesting that the agents weren’t learning reliably, almost as if their learning process was too random. We also ran the same test using the IPPO algorithm in a 4 marines versus 3 marines scenario and monitored the win rate:



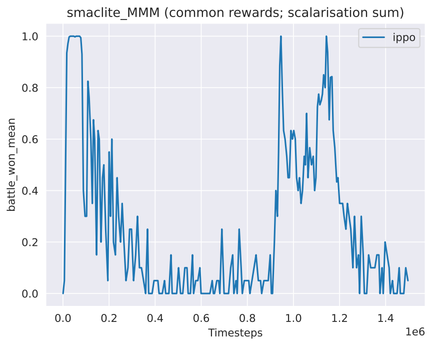


Figure 5.4: Default Reward in a 4 marines VS 3 marines scenario with IPPO algo

As we can see, the results with the IPPO algorithm are also unstable, the win rate fluctuates significantly. It's important to note that these training runs did not involve any interaction with the LLM and used only the default reward function. This further supports the conclusion that EPyMARL is not behaving as expected in our setup.

Conclusion

The experiments conducted using EPyMARL with the default reward function in SMACLite showed that the results were not stable or consistent. The high variation in the win rate, episode length, and reward suggests that the framework may not be ideal for the use of LLM-generated reward functions. Although EPyMARL is the recommended framework for training agents in SMACLite, we found that it does not suit our needs well. In our case, EPyMARL produced unstable results even when using the default reward function. This made it difficult to run consistent experiments and highlighted the need for a different framework that supports faster and more reliable training (Mava), especially when running many experiments. In the next section, we examine the results of using the Mava framework with the default reward function to see if it produces more stable outcomes. If so, it would provide a more reliable baseline for comparing against our LLM-based reward training.

5.2 Mava Results

These results are made with a IPPO algo and a Sebulba architecture. Indeed, SMACLite is a competitive MARL setting where agents must not only optimize their own performance but also consider and outmaneuver other agents. Sebulba's strategy of anticipating and exploiting the flaws of others aligns well with the dynamics of SMACLite, where agents need to adapt quickly and act strategically.

5.2.1 Default Reward

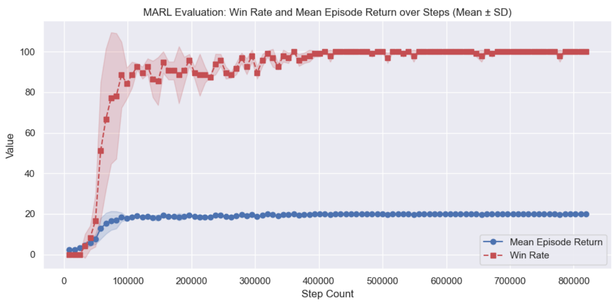


Figure 5.5: 4vs3 Win Rate And Mean Reward Default Reward

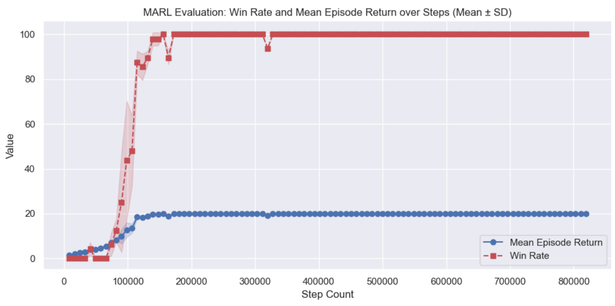


Figure 5.6: 3 vs 3 Win Rate And Mean Reward Default Reward

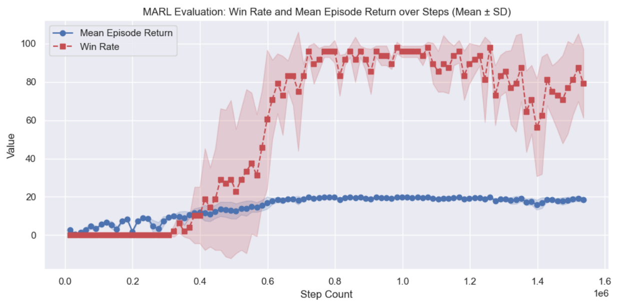


Figure 5.7: 10 vs 11 Win Rate And Mean Reward Default Reward

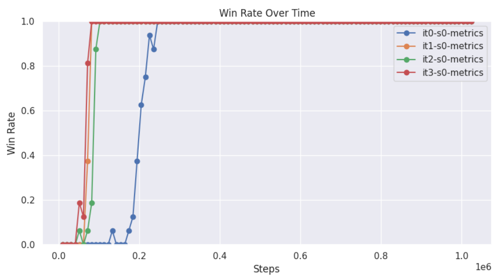
In all scenarios with the default reward, agents show clear learning progress.

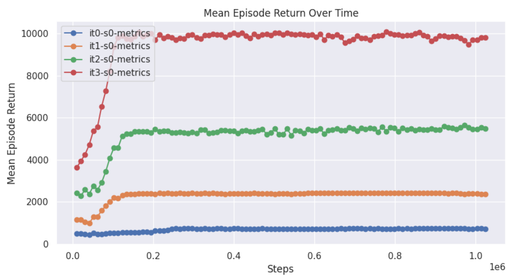
- 4v3 scenario: Wins begin to appear before 50,000 timesteps, with steady reward growth indicating increasing winning.
- 3v3 scenario: Wins start showing up slightly later, around 75,000 timesteps.
- 10 vs 11 scenario: Wins start to go up around 350,000 timesteps

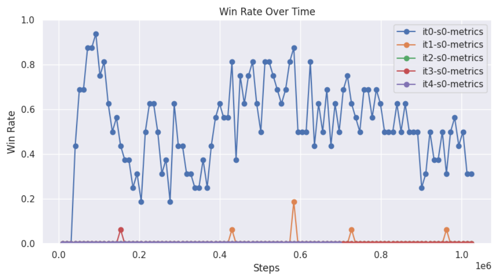
This demonstrates consistent behavior, as the win rate increases steadily with the number of timesteps and shows minimal fluctuation in simpler scenarios. In contrast, in more challenging setups; such as the 10 vs 11 scenario, agents begin to achieve wins later, and the results exhibit greater variability. This increased fluctuation is due to the inherent randomness of RL training and the complexity of the task. Overall, these observations confirm that the default human-designed reward is well suited for the environment, and that Mava's algorithm provides stable and reliable performance across varying levels of difficulty.

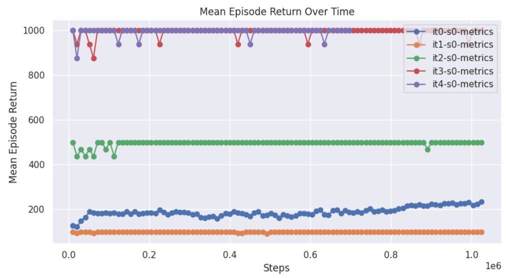
5.2.2 LLM Reward

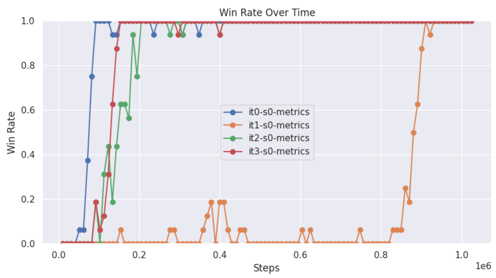
gpt-4o Focus Fire

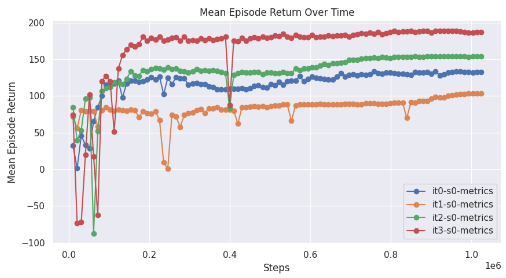












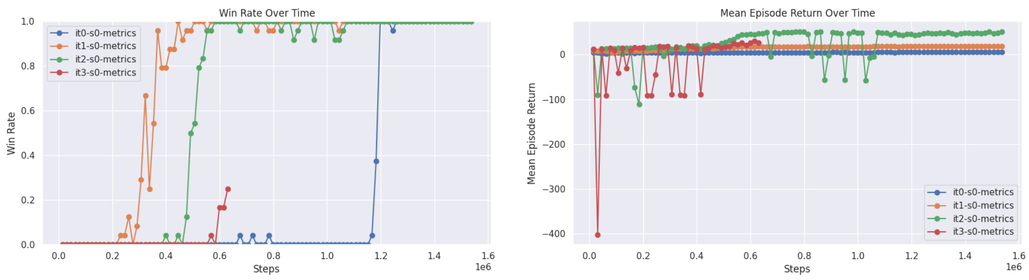


Figure 5.8: Gpt-4o Reward Focus Fire in a 4 marines VS 3 marines scenario with IPPO algo

The LLM generally designs effective reward functions by the end of each Eureka process, with most runs reaching a 100% win rate. In the 4 vs 3 scenario, where alies outnumber enemies, experiment 1 shows steady improvement across iterations: later reward functions learn more quickly to eliminate foes, and by the final iteration of experiment 1 and the first of experiment 3 they begin winning at roughly 50,000 timesteps, matching the Default Reward. Thus, in this setting the LLM can craft reward functions as effective as the human-designed one, even if simpler. Experiment 4 takes longer to excel but still yields reward functions that ultimately hit 100% wins. By contrast, experiment 2 fares poorly: its first iteration yields unstable learning, and later iterations offer no gains, illustrating that prompt iteration is not always beneficial. Still, with 3 of 4 experiments succeeding, GPT-4o proves largely capable of designing reward functions for this scenario

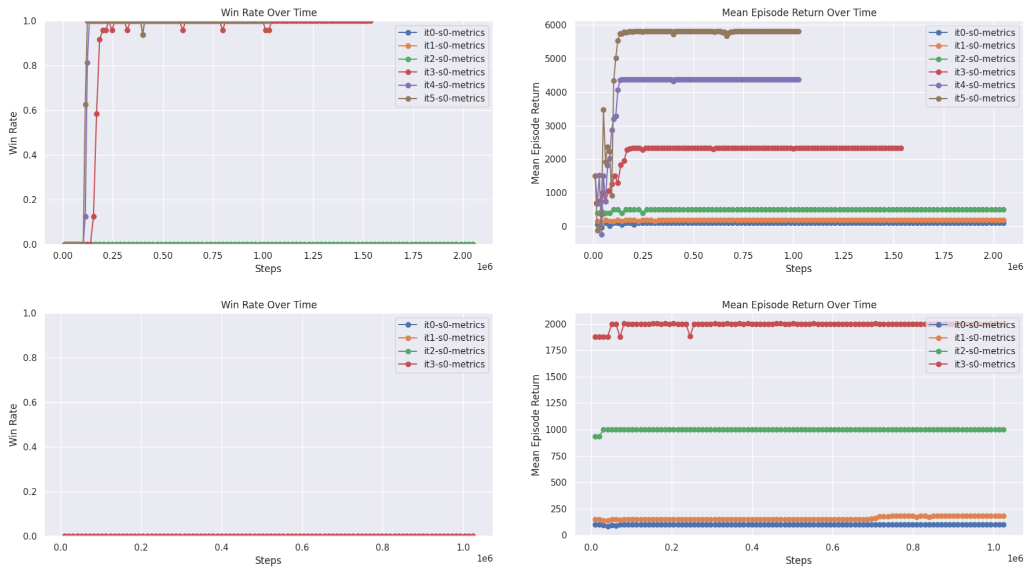
Regarding the mean reward of the first experiment, we can see that it increases with each iteration, suggesting that the LLM is adjusting the reward scale to better guide learning. In the final iteration, the mean reward stabilizes around 10,000 which is much higher than the Default reward, which rarely exceeds 20. This highlights that what matters most isn't the absolute reward value, but whether the reward function consistently reflects better or worse actions. As long as the agent can distinguish improvement e.g., 4,000 vs 10,000 it can learn effectively. That was for the first experiment, but we can see that the reward scale changes a lot across the experiments. In the third experiment, all the rewards from the different reward functions stay below 200, and sometimes even go negative. In Last experiment, we often see negative rewards and small positive rewards. The reward scale is part of the design of the reward function, but as explained earlier, the most important thing is being able to learn and distinguish good actions from bad ones. This is a good example that shows different reward scales can still work to complete the same task successfully.

Finally, compared to the Default reward function, the best reward function starts winning at about the same time. Like the Default reward, the LLM's reward also focuses on reducing enemy health. Even though the prompt says to use a Focus Fire strategy without explaining how, the LLM still figures out how to create a reward that helps the agent beat the enemies by checking the enemies health.

```
# Update the reward components
for agent_index, obs in enumerate(observation):
    enemies_info = obs[4:4 + 5*3]

    # Compute enemy health sum
    total_enemy_health = sum(enemies_info[i+4] for i in range(0, len(enemies_info), 5))

    # Reward component for reducing enemy health
    previous_enemy_health = reward_components.get("prev_total_enemy_health", total_enemy_health)
    health_reduction_reward = previous_enemy_health - total_enemy_health

    # Stronger focus on reducing enemy health
    total_reward += health_reduction_reward * 1000
    reward_components["prev_total_enemy_health"] = total_enemy_health

    # Focus fire reward
    if action[agent_index] in range(6, 9):
        target_enemy = action[agent_index] - 6
        if enemies_info[target_enemy * 5 + 4] > 0:  # Target is alive
            focus_fire_bonus += 100 *
            # Increase reward for focus fire
            action.count(action[agent_index])
```

Figure 5.9: Code from experiment 1 and iteration 3

In this section of the code, we observe how the LLM-generated reward function updates the reward components. A significant reward is granted for reducing enemy health. Additionally, the final part of the code implements the focus fire strategy, where a bonus is awarded if multiple allies target the same enemy. As in the Default Reward the reward Function of the LLM give strong reward when damaging enemies, this show that both reward have similitude with each other.



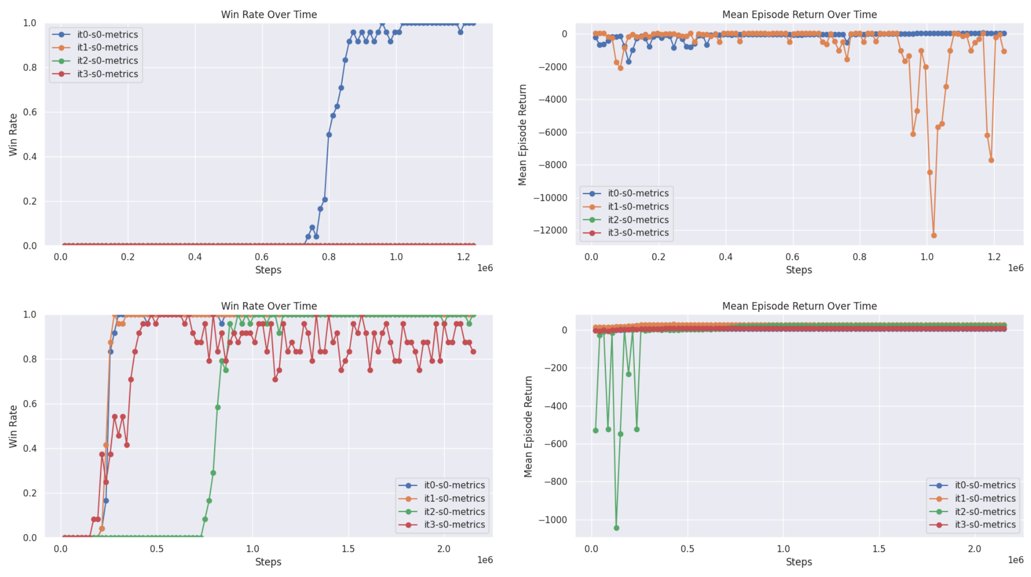


Figure 5.10: Gpt-4o Reward Focus Fire in a 3 marines VS 3 marines scenario with IPPO algo

In the tougher 3 vs 3 scenario, agents lose their numerical advantage. Experiment 1 shows the LLM's first three reward functions failing (win rate 0%), but the last three eventually hit 100%, indicating Eureka finally corrected the reward. The fastest learner and the default reward both start winning near 100,000 timesteps and soon lock at 100%, proving GPT-4o can match the human-made reward here. Experiments 2 and 3 were weaker. In experiment 2 no valid reward emerged; scaling tweaks visible in the mean-reward graph still left the agent unable to win. Experiment 3's first iteration did reach 100% slowly, but later iterations produced invalid rewards that could not teach the agent. Experiment 4 partly repeated experiment 1: the first iteration's reward was already strong, so further Eureka passes changed little. The last iteration did achieve wins earlier, though it never restored a steady 100% rate.

We can conclude that this model is sometimes capable of producing reward functions achieving effective learning results for this scenario, but it doesn't work consistently. In fact, the reproducibility of the experiment appears to be fifty fifty, with only 2 out of 4 experiments yielding successful outcomes.

```
focus_fire_bonus = 150.0  # Increased to emphasize focus fire
# Focus fire bonus: reward more if the same enemy is targeted by multiple agents
enemy_target_counts = np.zeros(num_enemies)
for a in action:
    if 6 <= a <= 8:
        target_enemy = a - 6
        enemy_target_counts[target_enemy] += 1
reward += np.sum(enemy_target_counts[enemy_target_counts > 1]) * focus_fire_bonus
```

Figure 5.11: Code from experiment 1 and iteration 5

This is how the focus fire bonus is implemented in the last iteration of the first experiment with the 3 vs 3 setup. It works well the code gives 150 points for each enemy that's targeted by more than one ally at the same time. When the strategy is clearly explained, GPT-4o is able to accurately reflect the Focus Fire strategy in the reward function for this scenario.

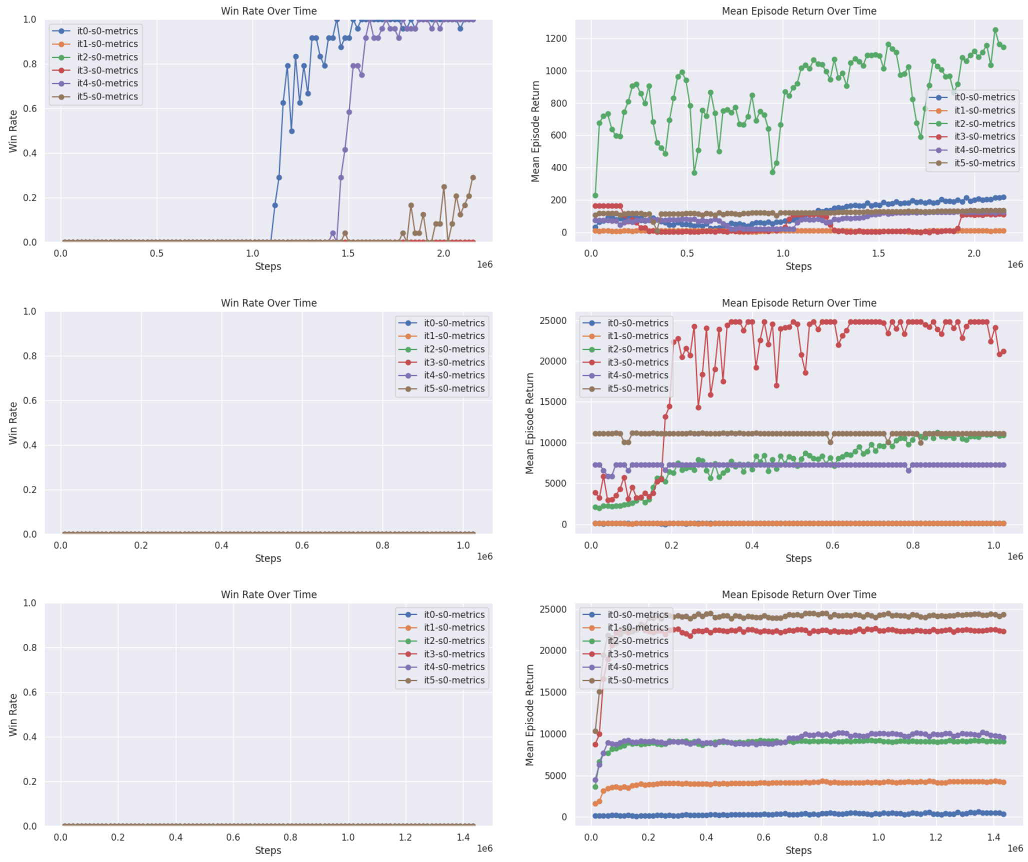


Figure 5.12: Gpt-4o Reward Focus Fire in a 10 marines VS 11 marines scenario with IPPO algo

For this scenario, the LMM faces more difficulties. First, it has a numerical disadvantage, and the number of agents to deal with is more complex. Indeed, the observation space is larger and may be harder for the LLM to understand, although the principle remains the same as before. We observe that for the first experiment, the best iteration was the first one, which is strange since we expect that the quality of the reward functions improved with the Eureka iteration. Here, however, this is not the case. The first iteration performs the best, and then the results decline. Iterations 1, 2, and 3 all have a win rate of zero, while the last two still achieve a win rate but learn more slowly than iteration 0. The other experiments 2 and 3 didn't produce any interesting reward function. When comparing the only interesting result to the Default reward training, we see that the first iteration takes much longer to achieve the goal 1,100,000 timesteps compared to 350,000 with the default reward. In this scenario, we can say that the LLM fails to generate reward functions that perform as well as the default one.

In the first experiment, the mean reward stays below 200 in all iterations except for iteration 2, which didn't lead to any success. Iteration 1 shows the biggest improvement over time, which is a good sign that the agents are learning. The other iterations don't show this pattern. In experiments 2 and 3, the mean reward increases early for most reward functions, but this doesn't help the agents learn how to win. This suggests the reward functions aren't well designed, agents are still maximizing reward, but without reaching the goal (Reward Hacking). We will see what component of the reward function could lead to such behavior.

Bellow is the code of the reward function from the iteration 2 of experiment 2. For this reward function, we see that the mean reward increases over time from below 5,000 to over 10,000 after 1,000,000 timesteps without the agent actually reaching the goal.

```python
num_agents = 10
num_enemies = 11
max_steps = 500

# Initialize reward components
reward_focus_fire = 0.0
reward_enemy_elimination = 0.0
reward_time_penalty = 0.0
reward_threshold_bonus = 0.0
total_reward = 0.0

# Track enemies' health and attacks
current_enemy_health = np.zeros(num_enemies)
attacks_on_enemies = np.zeros(num_enemies)

# Parse observations to determine health and actions directed at enemies
for agent_idx in range(num_agents):
    obs = observation[agent_idx]

    for enemy_idx in range(num_enemies):
        base_idx = 4 + 5 * enemy_idx
        enemy_health = obs[base_idx + 4]
        if enemy_health > 0:
            current_enemy_health[enemy_idx] = enemy_health
        if 6 + enemy_idx == action[agent_idx] and enemy_health > 0:
            attacks_on_enemies[enemy_idx] += 1

# Encouragement for focus firing: reward for concentrating attacks on the same enemy
reward_focus_fire += np.sum(attacks_on_enemies * (attacks_on_enemies - 1) / num_agents)

# Reward for significant enemy health reduction thresholds and total health reduction
previous_enemy_health = np.array([reward_components.get(f'enemy_{i}_health', 1.0) for i in range(num_enemies)])
enemy_health_lost = previous_enemy_health - current_enemy_health
reward_enemy_elimination += np.sum(enemy_health_lost)

# Bonus for reducing any enemy health below critical thresholds
critical_health_threshold = 0.2
for i in range(num_enemies):
    if current_enemy_health[i] < critical_health_threshold and previous_enemy_health[i] >= critical_health_threshold:
        reward_threshold_bonus += 2.0  # reward extra for reducing below critical threshold

# Time penalty to encourage faster completion
if step >= max_steps:
    reward_time_penalty -= 1.0

# Sum all reward components
total_reward = (
    reward_focus_fire * 0.1 +  # Scale focus fire rewards
    reward_enemy_elimination * 1.0 +
    reward_threshold_bonus * 0.5 +
    reward_time_penalty
)

# Record the current health for the next step comparison
for i in range(num_enemies):
    reward_components[f'enemy_{i}_health'] = current_enemy_health[i]

# Return total reward as a NumPy array repeated for each agent (assuming 10 agents)
return np.full(num_agents, total_reward, dtype=np.float32), reward_components
```

Figure 5.13: Code from experiment 2 and iteration 2

The reward function, as currently designed, contains several elements that allow agents to accumulate high rewards without actually achieving the intended goal which is defeating all enemies. The main problem lies in the way rewards are assigned based on enemy health reduction rather than on enemy elimination. The function gives a significant reward for any decrease in enemy health, which encourages the agent to spread damage across multiple enemies instead of focusing on eliminating them. This means that an agent can maximize its reward by simply lowering the health of many enemies a little, rather than fully defeating any of them.

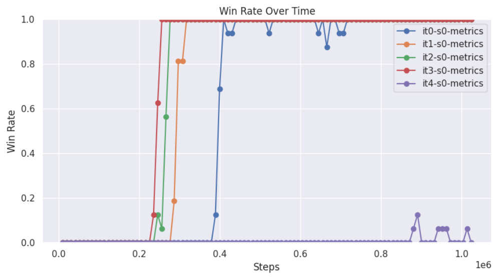
Additionally, while the function includes a focus fire component meant to encourage agents to concentrate their attacks, this part of the reward is heavily down scaled (multiplied by 0.1), making it too weak to influence the agent's behavior compared to the general damage reward. As a result, the agent may ignore this signal entirely.

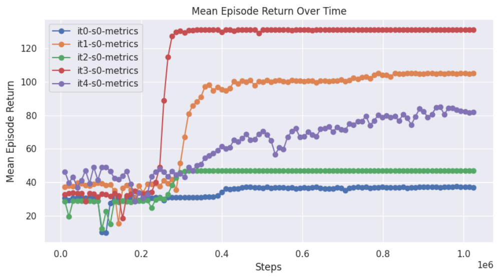
There is also a reward bonus given when an enemy's health drops below a critical threshold (e.g., 0.2), but this is only triggered once, regardless of whether the enemy is later finished off or not. This encourages the agent to repeatedly push enemies just below the threshold for the bonus, again without needing to defeat them, which is a classic form of reward hacking maximizing reward signals without achieving the task's true goal.

Furthermore, the function lacks any explicit reward for actually winning, there is no sparse reward such as eliminating all enemies even if it is suggested to use them in the initial prompt. Without this, there's no clear signal guiding the agent to complete the overarching objective.

Together, these design choices lead the agent to optimize for easy-to-reach sub goals (like small health reductions or threshold bonuses) instead of focusing on the more complex and meaningful goal of winning the fight. The remaining reward functions share the same design flaws in this experiment, which explains the consistently zero win rate.

gpt-4o-latest Focus Fire





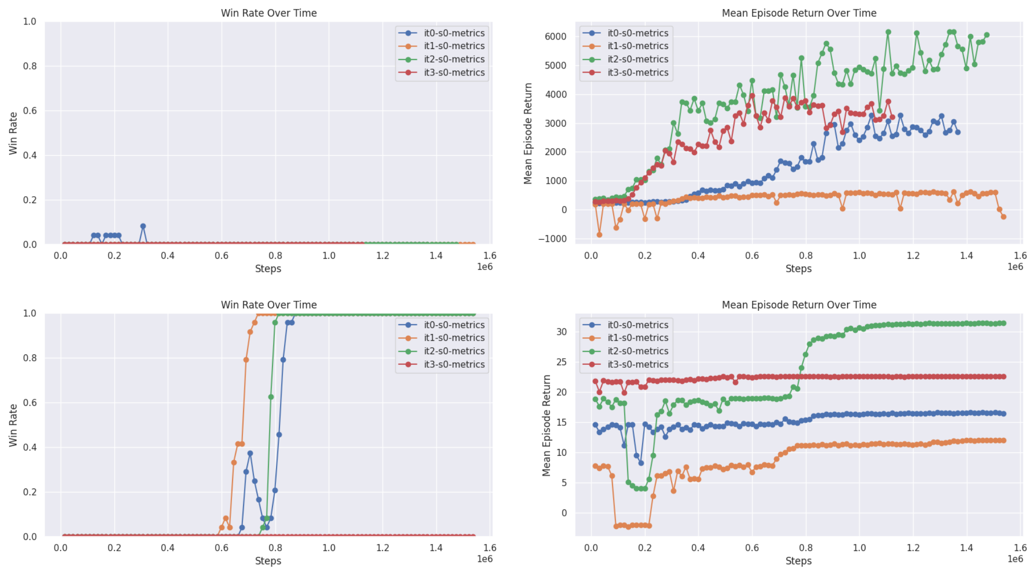
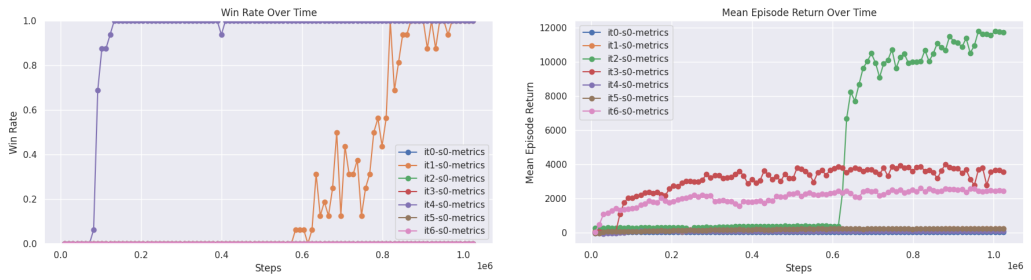


Figure 5.14: Gpt-4o-latest Focus Fire Reward in a 4 marines VS 3 marines scenario with IPPO algo

We can see in this first scenario that gpt-4o-latest also produces rewards functions that achieves a 100% win rate at some point, especially in experiment 1 and 3. However, the model learns more slowly compared to when the reward is provided by the fixed gpt-4o version, best reward only starts to win after 200,000 timesteps against 50,000 for the best gpt-4o reward. For the first experiment, after each iteration, the reward function improves, as it takes less time to learn except in the final iteration, which appears to be completely incorrect. The Second experiment is a fail and the third produce reward function that achieve a 100% win rate but slower than in the first experiment. The mean reward in the second experiment clearly shows a case of reward hacking, similar to what was observed earlier.



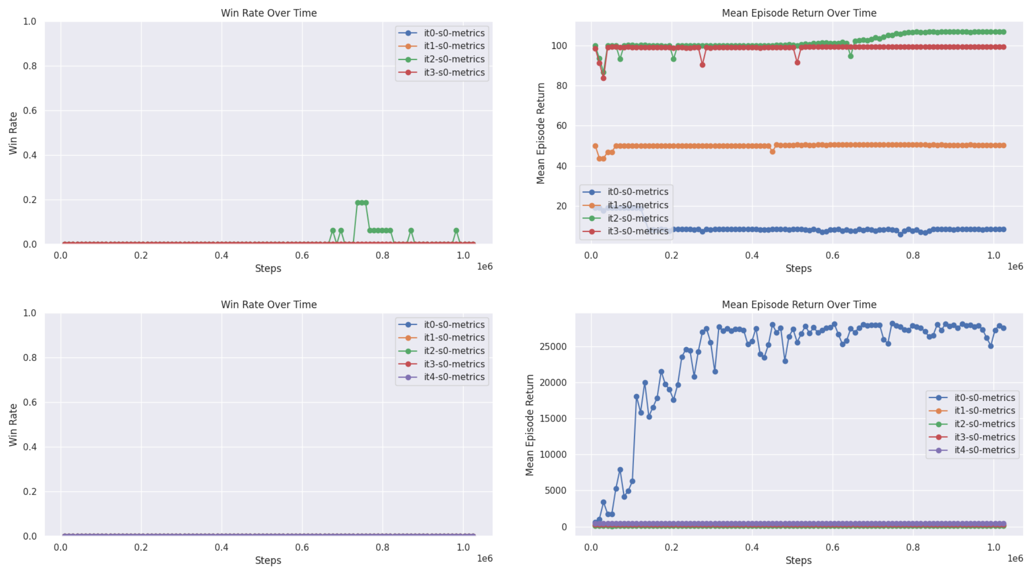


Figure 5.15: Gpt-4o-latest Focus Fire Reward in a 3 marines VS 3 marines scenario with IPPO algo

In the 3 marines versus 3 marines scenario, gpt-4o-latest seems to struggle more than the older gpt-4o version. In fact, it only manages to produce a well designed reward function by the fifth iteration of the first experiment, all other ones are not well suited for the task. The reward function from the fifth iteration gives smaller rewards, as shown in the second graph of the first experiment, but it still manages to successfully complete the task. The other two experiments failed. In the last experiment, the mean reward function of the first iteration increase a lot but still achieve a constant win rate of zero:

```
# Reward for reducing enemy health (aggressive focus on enemy)
delta_enemy_health = prev_enemy_health - current_enemy_health
reward_enemy_dmg = delta_enemy_health * 5.0  # aggressive weight

focus_fire_reward = 0.0
for count in attack_counts:
    if count >= 2:
        focus_fire_reward += count * 0.5  # bonus for coordinated attacks

enemies_killed = prev_enemy_units - new_enemy_units
kill_reward = enemies_killed * 15.0

# Success/failure rewards
terminal_reward = 0.0
if done:
    if new_enemy_units == 0:
        terminal_reward += 50.0  # won
    elif new_ally_units == 0:
        terminal_reward -= 50.0  # lost
    elif step >= 500:
        terminal_reward -= 20.0  # timed out
```

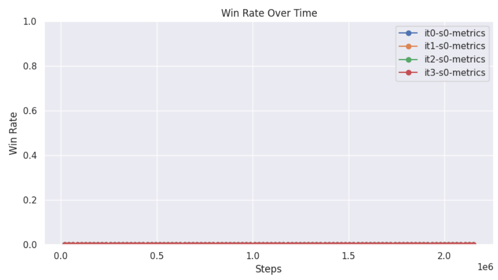
Figure 5.16: PArts of Code from experiment 3 iteration 0

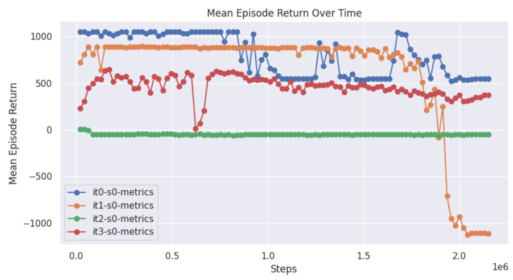
One of the most significant issues lies in the aggressive weighting of the reward for reducing enemy health. The function multiplies the change in enemy health by a factor of 5, giving strong, dense rewards for any damage dealt. This emphasis on health reduction, rather than elimination, risks encouraging agents to distribute damage across multiple enemies rather than focusing on killing them. In contrast, the reward for actually eliminating an enemy is only 15 per kill, which may not be high enough to outweigh the easier, continuous rewards gained from merely lowering health across multiple targets. As a result, agents may learn to "farm" partial damage for high reward instead of pursuing the more meaningful objective of winning the fight.

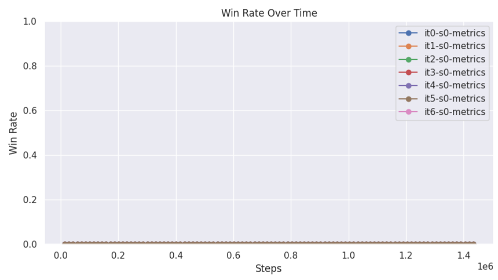
Although there is a bonus for focus fire when multiple agents attack the same target, the incentive is relatively small (0.5 per coordinated attack) and might be overshadowed by the dominant health reduction reward. If the focus fire bonus is not scaled appropriately, it will fail to sufficiently shape coordinated behavior and leading agents to act independently or inefficiently.

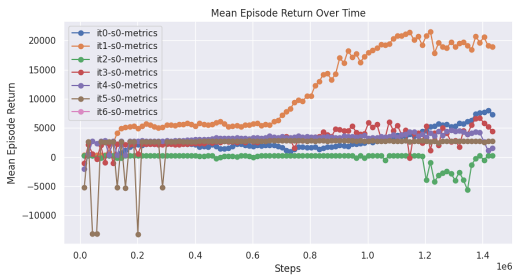
Finally, while the terminal rewards do provide clear success and failure signals: 50 for a win, -50 for a loss, they are sparse and only available at the end of an episode. Moreover, since the agent can still achieve high total rewards without ever winning, these terminal bonuses may not be sufficient to redirect the learning process toward the actual goal of consistent victory.

It's important to note that this is just the first iteration, and having scaling issues early on is acceptable. We expect these problems to be fixed in later iterations, but that didn't happen in the last experiment.









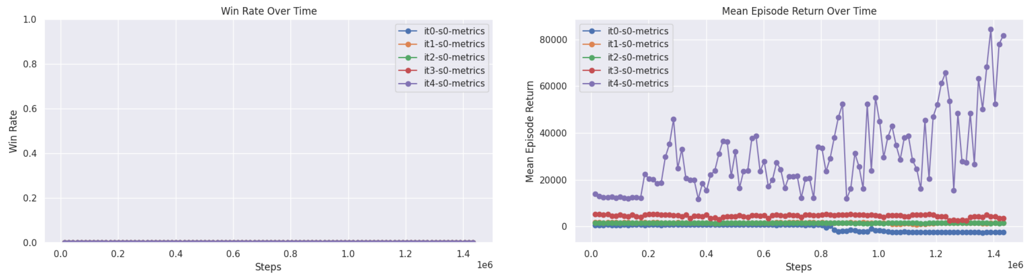


Figure 5.17: Gpt-4o-latest Focus Fire Reward in a 10 marines VS 11 marines scenario with IPPO algo

The results for the most difficult scenario are quite clear: the LLM did not produce any useful reward function. While the fixed gpt-4o version also struggles more in this scenario, it still manages to generate a reward function that eventually learns how to win slower than the default reward in his first experiment, but still valuable. In contrast, none of the reward functions from gpt-4o-latest are usable for this more difficult task. This might suggest that there is some regression in the latest updates of the model for the completion of our task which is generating well designed reward function for SMAClite. We should also keep in mind that even though the experiment was repeated three times, GPT-4o's success in this scenario in the first experiment could still be due to luck. So, we can't say with full certainty that GPT-4o-latest performs worse on this task than GPT-4o-2024-08-06.

It's important to note that different numbers of iterations were performed across experiments to determine if more iterations were needed to refine the reward function and make it better suited for the task. However, this did not affect the outcome.

```
# === Focus Fire Reward ===
focus_fire_reward = 0.0
focus_fire_kill_bonus = 0.0
for eid, count in targeted_enemies.items():
    if count >= 3:
        focus_fire_reward += (count - 2) * 5.0
        # Check if this enemy died this step
        enemy_idx = 4 + eid * 5
        enemy_health = observation[0][enemy_idx + 4]
        if enemy_health == 0.0:
            focus_fire_kill_bonus += 15.0
```

Figure 5.18: Code from experiment 2 iteration 4

The Focus Fire logic appears to be well implemented, but there's a scaling issue. The reward for focus fire is too small compared to the win reward. For example, winning gives 1500 points, while focus fire might only give around 15, which has very little influence. This means the agent isn't strongly encouraged to adopt the focus fire behavior. That explains one of the potential issues.

o3-mini Focus Fire

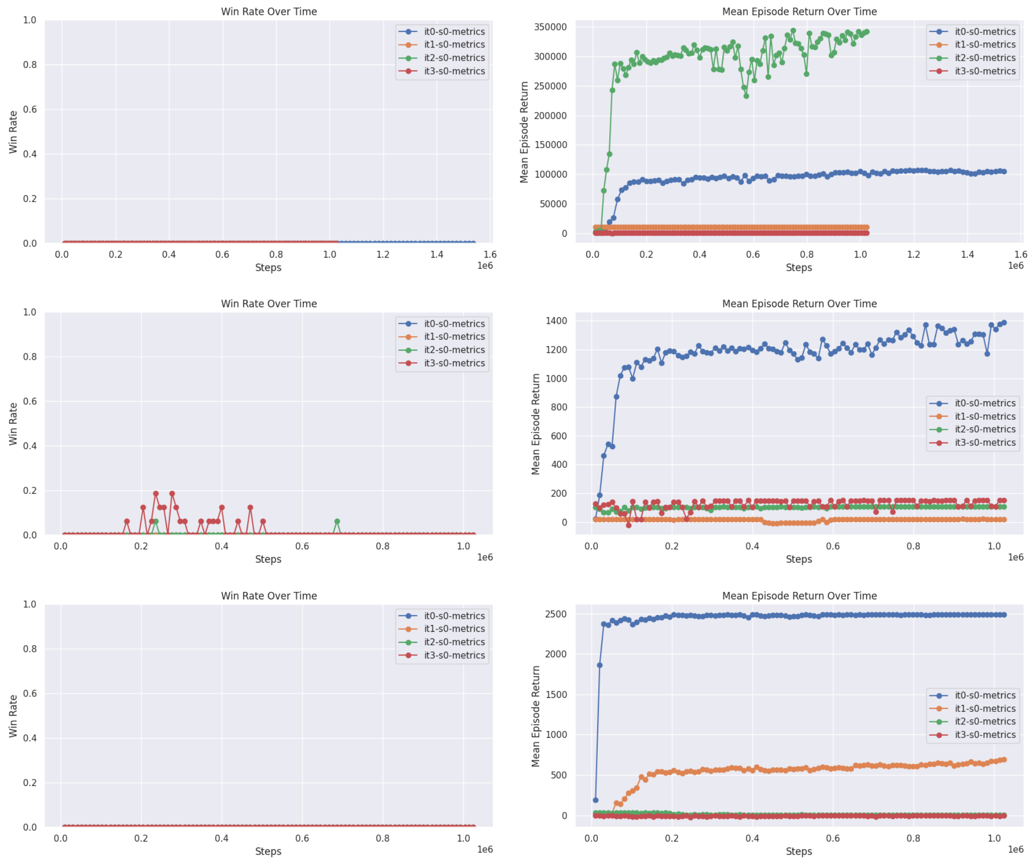


Figure 5.19: o3-mini Focus Fire Reward in a 4 marines VS 3 marines scenario with IPPO algo

For the easiest scenario, we can see that the O3-mini model fails to produce any well designed reward function for the task. Since it's the simplest task, such results doesn't give much confidence for its performance in the next more challenging scenarios.

The mean reward is higher for iterations 0 and 2, but this did not help the agents learn how to win. The mean reward is also extremely high, around 300,000, which might suggest a problem with the reward function design. In these iterations, the agents learned to maximize the reward without actually winning (Reward Hacking). If we look at the three mean reward graphs, we can see they show similar patterns. In all three experiments, the first iteration improves quickly at the beginning. However, the scale is different: in the first experiment, the mean reward levels off around 100,000 timesteps, in the second around 1,300, and in the third around 2,500. This might be because the mini model gives more consistent answers, with less variation between the same prompts.

Also, it’s worth noting that the first iteration of the first experiment ran 600,000 timesteps longer than the others. This was a small mistake, but it doesn’t affect the results. A training length of 1,000,000 timesteps is enough, since the Default Reward function starts winning after just 100,000 timesteps.

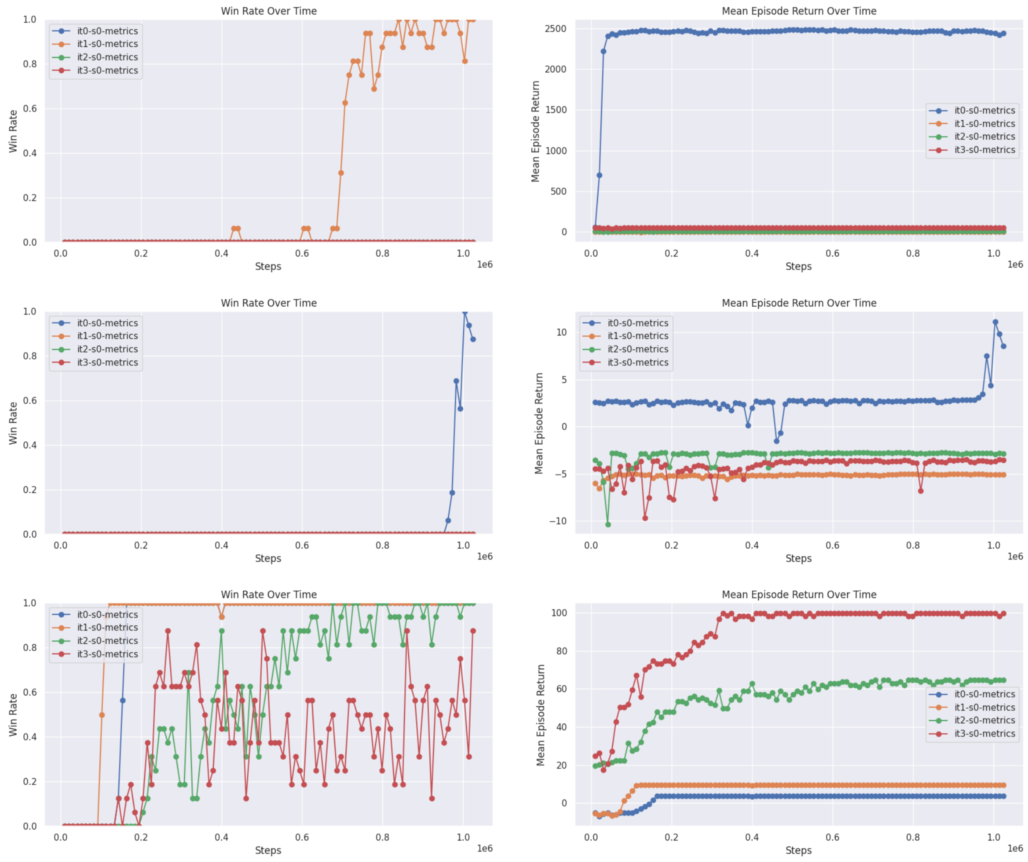


Figure 5.20: o3-mini Focus Fire Reward in a 3 marines VS 3 marines scenario with IPPO algo

In a more difficult scenario, O3-mini does manage at some point to produce a somewhat effective reward function for the task in the first experiment. However, the quality doesn’t match the reward functions generated by the GPT-4o models. Learning is slower, and there’s no clear improvement over time, it seems more like a lucky outcome rather than a sign of real understanding. The following iterations lose all previous progress, suggesting the model doesn’t recognize or correct the issues in its reward strategies.

In the second experiment, the first iteration only starts winning near the end of training (around 1,000,000 timesteps). We can see the mean reward increasing along with the win rate, but the agent learns how to win too late. Still, this shows that the reward function can tell the difference between winning and losing situations. The next iteration, however, shows a constant zero win rate.

In the last experiment, things are more interesting. The first iteration is already well designed and learns quickly just before 200,000 timesteps. The second iteration performs just as well as the Default reward, with wins starting around 100,000 timesteps, which is fast. The last two iterations do worse, learning more slowly and failing to produce stable behavior. In this case, just two "eureka" iterations were enough to get the best results. Asking the LLM to improve a reward function that's already close to optimal can actually make it worse.

```
focus_fire_scale = 1.0
# --------------------------------------------------
# 2. focus_fire Component
# --------------------------------------------------
# Calculate the number of unique enemy targets among attack actions.
attack_targets = []
for act in action:
    if 6 <= act <= 8:
        # Convert action to enemy index (0, 1, or 2)
        attack_targets.append(act - 6)
if attack_targets:
    current_focus_fire = float(len(set(attack_targets)))
else:
    # If no attack action was taken, assume worst-case
    current_focus_fire = 3.0
# Default previous focus criterion is 3 (worst coordination) if not provided.
prev_focus_fire = reward_components.get("focus_fire", 3.0)
focus_fire_reward = (prev_focus_fire - current_focus_fire) * focus_fire_scale
```

Figure 5.21: Code from experiment 1 and iteration 3

We can see that the implementation of the Focus Fire strategy is different that what we have seen so far with the other models. It checks how many unique enemies are being targeted: fewer targets mean better coordination. A reward is given when the number of unique targets decreases compared to the previous step. However the reward is quite small for Focus Fire which might explain why the agents struggle to learn how to win.

In conclusion, the results in a 4 vs 3 scenario are the worst seen so far, with no valuable reward function produced. We expected similar results in a more complex scenario, but this was not the case. In a 3 vs 3 scenario, O3-mini performed better than expected. In each experiment, it produced at least one reward function that successfully guided the agent to learn how to win. So, despite the challenges in the more complex setup, O3-mini showed that it can adapt and perform well in simpler environments, demonstrating its potential when the scenario is appropriately scaled.

O3-mini was also tested on the more difficult scenario (10 vs 11). However, it failed to produce any reward function that helped the agents learn how to win across the 3 experiments and 12 generated reward functions. Just like GPT-4o-latest, it didn't succeed in this task, so the graph results are not shown as they are not relevant.

gpt-4o Dancing

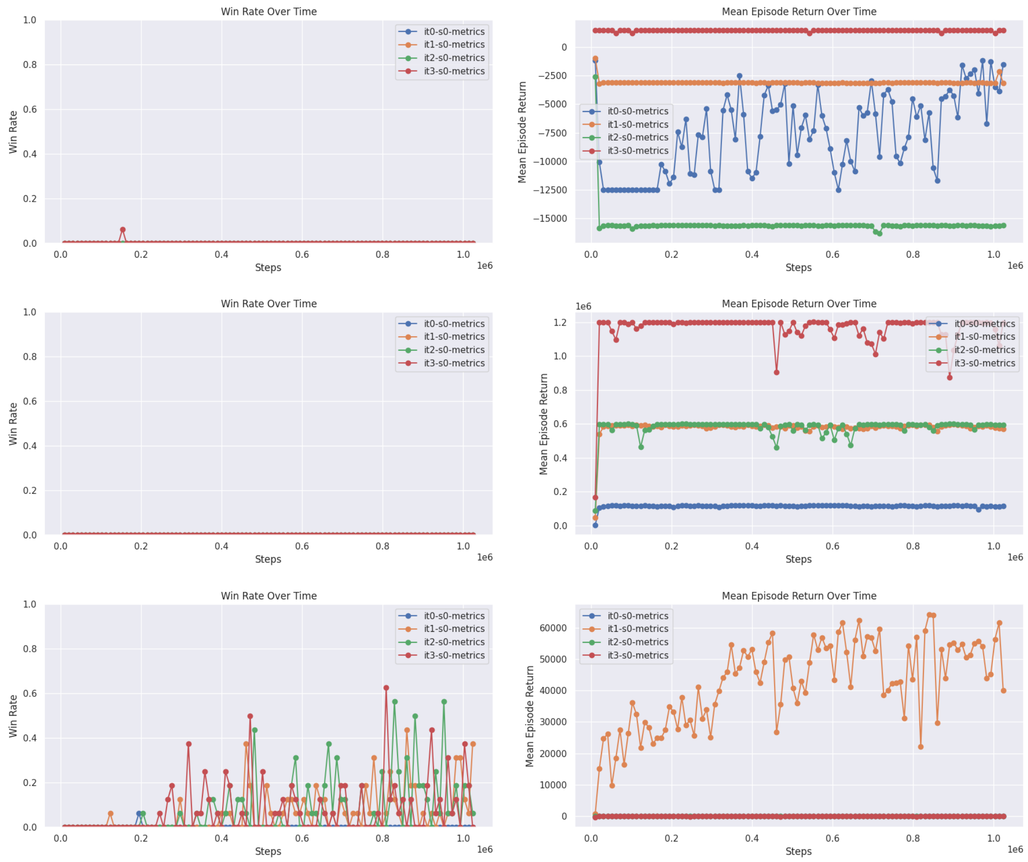


Figure 5.22: gpt-4o Dancing Reward in a 4 marines VS 3 marines scenario with IPPO algo

GPT-4o seems unable to produce a good reward when instructed to use the Dancing strategy. While it attempts to express this strategy in the reward function, it does so unsuccessfully. The model only follows the instructions given, without taking any initiative or adapting its approach on its own. In the last experiment, the model learned to win some episodes with iterations 1, 2, and 3, but in all cases, the win rate stayed below 70%. As a result, no improvement was observed across iterations, and the reward functions were not well designed to guide the agent towards achieving a 100% win rate.

```
# Introduce penalty for allies taking too much damage to encourage strategic retreats
ally_health_threshold = 0.3 * num_agents  # Threshold for retreat or dancing strategy
if current_ally_health < ally_health_threshold:
    total_reward -= 10
```

Figure 5.23: Experiment 1 and iteration 2 Dancing Implementation

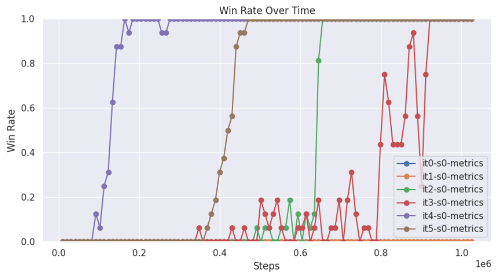
```
# Encourage 'dancing' by penalizing low ally health when under attack
```

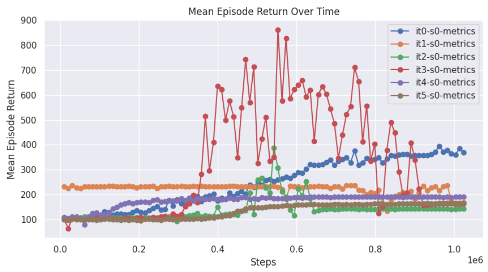
```
        if current_ally_health < previous_ally_health:
            total_reward -= 2 * (previous_ally_health - current_ally_health)
```

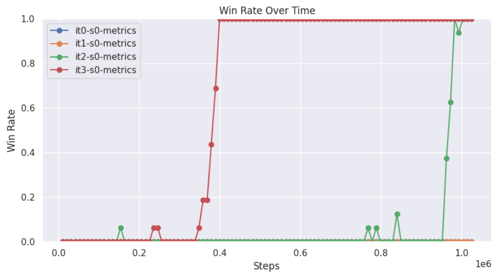
Figure 5.24: Experiment 1 and iteration 3 Dancing Implementation

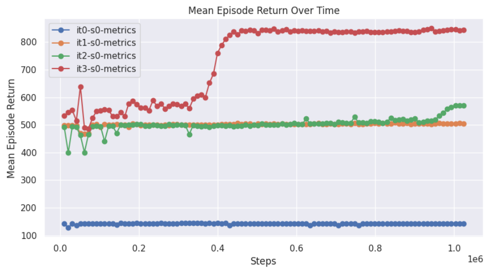
We can see that the only thing the LLM does to encourage the Dancing strategy is to add negative rewards based on the allies' health. The first implementation gives a -10 reward if all agents have health below a certain threshold. The second gives a negative reward if all agent's health keeps dropping over time. But this creates a problem, by punishing the agent whenever its health goes down, it ends up learning to run away from fights and never come back. That's not what the Dancing strategy is really about, it's supposed to mean retreat when low on health, then return to fight. Since nothing in the reward tells the agent to re-engage, it might just learn to run and hide to maximize his rewards, which is not what we want (see Reward hacking). This is a good example of the Double Bias effect: first, we bias the prompt by telling the LLM to use a Dancing strategy (which might not be the best for this scenario) and then the reward function it generates adds another bias during training of agents. One thing we can say for the defense of the LLM is that the Dancing strategy might not be the best strategy for this scenario and is more complex to implement with the information given to the LLM.

gpt-4o-latest Dancing









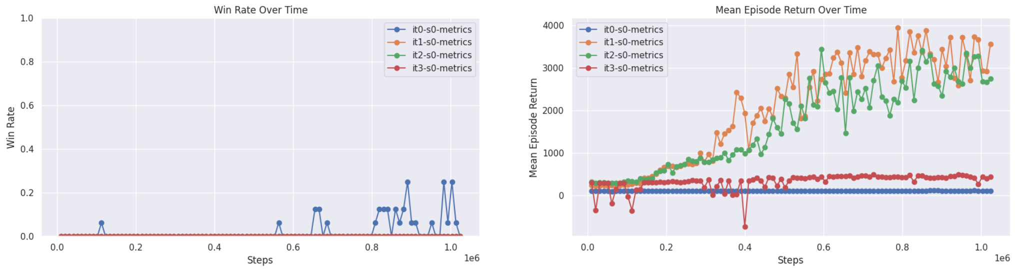


Figure 5.25: gpt-4o-latest Dancing Reward in a 4 marines VS 3 marines scenario with IPPO algo

We can see that the gpt-4o-latest model actually performs better with this prompt than gpt-4o. Unlike gpt-4o, which seems to get "lost," gpt-4o-latest finds a way on its own to generate a good reward function. What's interesting is that the prompt only mentions the "Dancing strategy," but the model also incorporates a "focus fire" strategy, despite no explicit mention of it in the prompt. This shows that gpt-4o-latest is able to take initiative, whereas gpt-4o seems confused when no specific strategy is provided. However, the results are not as strong as when the prompt explicitly instructs the model to use the focus fire strategy. One important point to note is that the focus fire strategy appears to be very effective in these scenarios and is sufficient to win, while the dancing strategy may be harder to express in the reward function and may not perform as well as focus fire. So the win rate probably come from the influence of Focus Fire strategy rather than the Dancing strategy.

```
# --- Focus fire bonus ---
focus_bonus = 0.0
for count in target_counts.values():
    if count > 1:
        focus_bonus += (count - 1) * 3.0

# --- Dancing bonus (low hp agent that moves instead of attacking)
dancing_bonus = 0.0
for i in range(num_agents):
    hp = observation[i][-1]
    act = action[i]
    if hp < 0.3:
        if act in [2, 3, 4, 5]:  # moves
            dancing_bonus += 1.0
        elif 6 <= act <= 8:  # attack while low HP
            dancing_bonus -= 0.5
```

Figure 5.26: Experiment 1 iteration 4 Micro Strategies Implementation

In the code above, we can see just like we mentioned earlier that gpt-4o-latest decided to introduce the Focus Fire strategy on its own, even though it wasn't mentioned in the prompt. Also, the way it implemented the Dancing strategy it's not what we want. It checks if an agent's HP drops below 0.3; in that case, the agent should move instead of attacking. If it moves, it gets a positive reward of 1; if it attacks, it gets a penalty of -0.5. But, the direction of the movement isn't checked. So if the agent moves towards the enemy, that's clearly not what we want but the reward still counts it as a positive move. It's complicated to check but not impossible since the LLM does have access to the enemies's normalized positions and could use that to judge whether the agent is actually retreating. In summary, this implementation of the dancing strategy tells agents with health below 0.3 to move, but it doesn't provide any direction for where to go or instruct them to return afterward, it simply tells them to move.

o3-mini Dancing

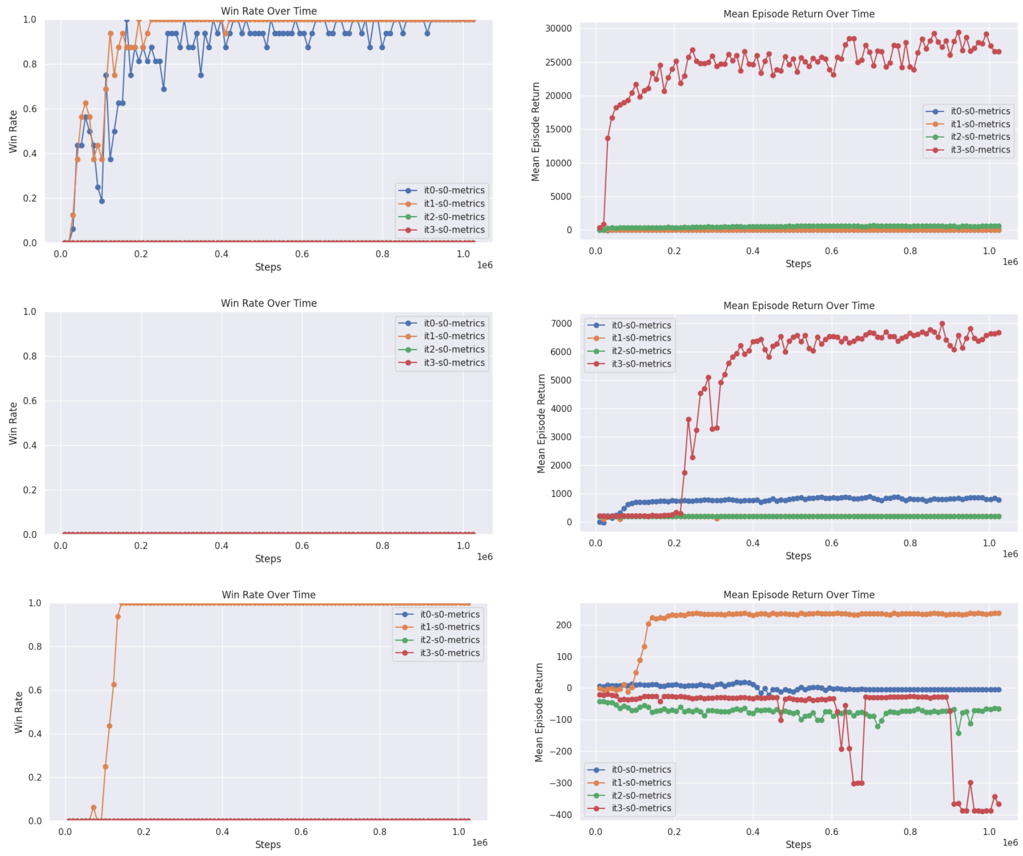


Figure 5.27: o3-mini Dancing Reward in a 4 marines VS 3 marines scenario with IPPO algo

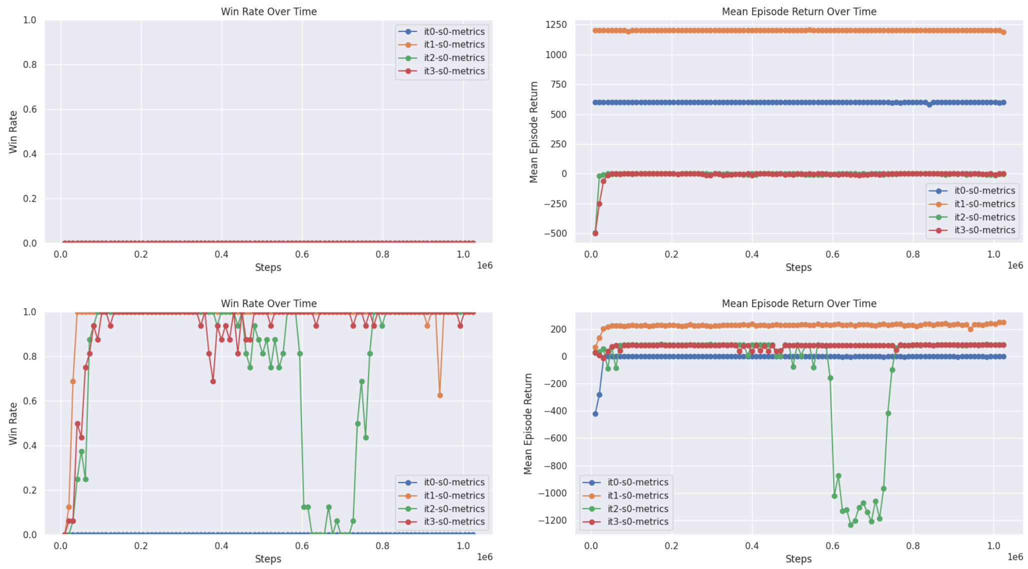
We can see that the first experiment is a success. o3-mini immediately generates a reward function that achieves a 100% win rate, and the following iteration slightly improves the stability of the learning. However, the last two iterations fail to accomplish the task. The second experiment did not produce any effective reward function, while the final experiment generated only one good reward function, all the others failed to achieve the task.

```
# Determine ideal retreat action if any enemy is attackable.
ideal_retreat = None
if count > 0:
    # Average enemy relative position.
    avg_dx = total_dx / count
    avg_dy = total_dy / count
    # The ideal retreat is in the opposite direction of the aggregate enemy position.
    if abs(avg_dx) >= abs(avg_dy):
        # Retreat horizontally.
        if avg_dx > 0:
            ideal_retreat = 5  # retreat west if enemy is to the east
        elif avg_dx < 0:
            ideal_retreat = 4  # retreat east if enemy is to the west
    else:
        # Retreat vertically.
        if avg_dy > 0:
            ideal_retreat = 3  # retreat south if enemy is to the north
        elif avg_dy < 0:
            ideal_retreat = 2  # retreat north if enemy is to the south

# If an ideal retreat direction is determined and the unit chooses that movement,
# reward the agent for "dancing". Also add a bonus if the unit had recently taken damage.
if ideal_retreat is not None:
    if action[agent_idx] == ideal_retreat:
        dancing_bonus += 0.1
        if (prev_health - current_health) > 0:
            dancing_bonus += 0.05
```

Figure 5.28: Experiment 1 iteration 1 Micro Strategies Implementation O3-mini

In this case, the o3-mini model, when prompted to implement a "dancing" strategy, was the first among the tested models to meaningfully incorporate enemy positioning and agent retreat behavior into the reward function. Unlike GPT-4o and GPT-4o-latest, which generated more basic reward functions that did not truly account for tactical retreats, o3-mini introduced logic that allows agents to move away from enemies based on their relative positions, an essential component of a proper dancing micro-strategy. The interesting part of the code calculates the average relative position of nearby attackable enemies (avg_dx, avg_dy) and determines an ideal retreat direction opposite to where the enemies are. For example, if enemies are mostly east, the agent is encouraged to move west. This could help explain why o3-mini performed well during the initial iterations of the first experiment.

No Micro Strategies Given to the LLM



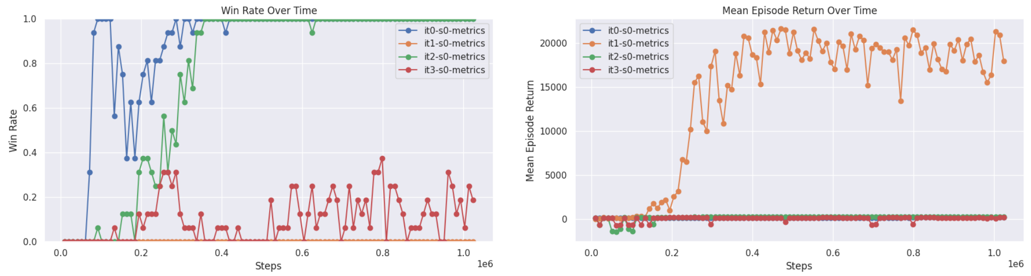


Figure 5.29: gpt-4o Reward in a 4 marines VS 3 marines scenario with IPPO algo and no micro Strategies given

We can see that the first experiment didn't produce any reward function that helped the agents learn how to win. The mean reward stays mostly constant across most iterations.

The second experiment has interesting results. The first iteration produce a constant zero win rate, then the second quickly learn and start to have win very quickly around 20,000 timesteps which is better than the default reward that start learning around 50,000 timesteps. The next iteration try to improve the reward but this is very hard since this is almost optimal. We will look at the code of the reward function from the iteration 1 to see what kind of strategies it implement itself without any guidance in the strategies to use in the prompt.

```
# Calculate rewards based on observations and actions
for agent_obs, agent_action in zip(observation, action):
    # Extract enemy information from observation
    enemies_info = agent_obs[4:19]  # 3 enemies * 5 attributes each

    # Reward for reducing enemy health
    enemy_health_component = 0.0
    for i in range(0, len(enemies_info), 5):
        attackable = enemies_info[i]
        enemy_health = enemies_info[i + 4]
        if attackable:
            previous_health = reward_components.get(f'enemy_health_{i}', enemy_health)
            enemy_health_component += previous_health - enemy_health
            reward_components[f'enemy_health_{i}'] = enemy_health

    reward_components['enemy_health'] += enemy_health_component

    # Encourage aggressive actions, particularly attacking
    if 6 <= agent_action <= 8:  # Attack actions are from 6 to 8
        reward_components['aggression_bonus'] += 2.0
```

Figure 5.30: Code from experiment 2 iteration 1

This reward function uses a very simple but effective strategy that focuses on two key elements:

1. Enemy health reduction: It gives a positive reward whenever an enemy's health decreases. For each agent, the function tracks the health of every enemy and adds to the reward whenever the current health is lower than the previous one. This directly encourages the agent to attack enemies and continue until they are defeated.
2. Aggressive behavior bonus: It rewards the agent with a fixed bonus (+2.0) whenever it chooses an attack action (actions 6 to 8). This pushes the agent to favor attacking over other possible behaviors like moving or idling.

This strategy works well in this scenario mainly because of its simplicity. A more advanced tactic like Focus Fire isn't required here, as the agents already have a numerical advantage. However, in more complex situations, this approach might not be sufficient. Interestingly, even though the reward function doesn't include any sparse rewards, the agents still manage to learn how to win quickly.

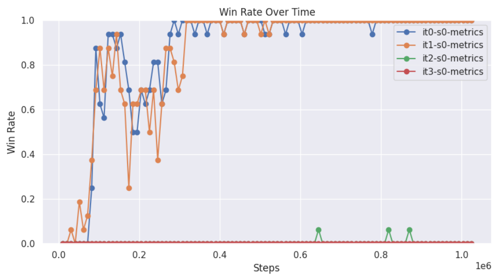
In the last experiment, only iterations 0 and 2 showed promising results, while the others either led to a constant zero win rate or unstable outcomes.

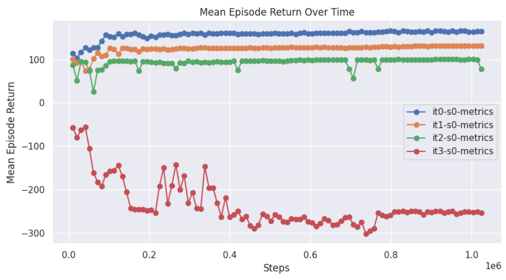
```python
# Check if episode is done
if done:
    if total_enemy_health == 0:
        reward_components['episode_end'] = 100.0   # Bonus for winning
    else:
        reward_components['episode_end'] = -50.0   # Penalty for losing
total_reward += reward_components['episode_end']
```

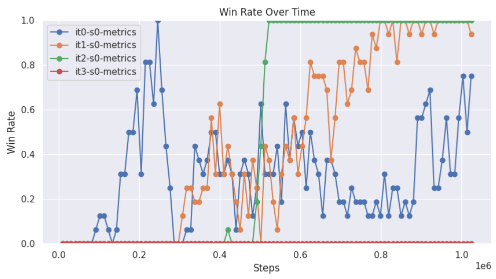
Figure 5.31: Code from experiment 3 and iteration 0 Sparse Reward

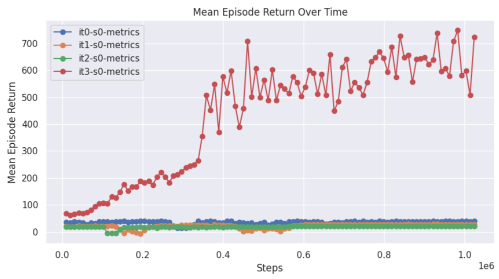
In the first iteration of this experiment, the reward function contains a sparse reward implementation, unlike iteration 2 in the previous experiment.This shows that even with sparse rewards, the agent can learn effectively when the signal aligns well with the task.

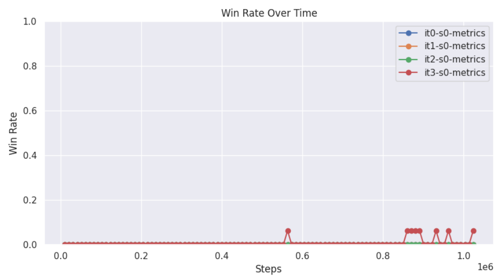
To conclude on the GPT-4o results without a given strategy, we see that it implements basic behaviors without mentioning known micro-strategies like Focus Fire or Dancing. It keeps things simple, which works well in this simple scenario, but may not be enough in more complex situations.











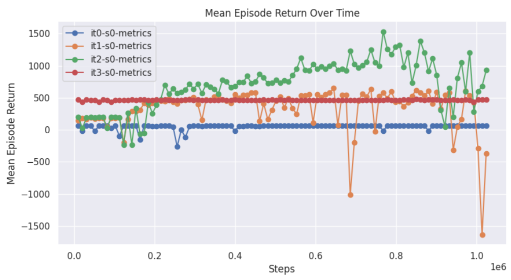


Figure 5.32: gpt-4o-latest Reward in a 4 marines VS 3 marines scenario with IPPO algo and no micro Strategies given

In the first experiment, we observe that the first two iterations produce reward functions that achieve a 100% win rate quickly, similar to the Focus Fire prompt. However, the last two iterations didn't result in any interesting reward functions. The learning in the first two iterations is unstable initially but stabilizes around 400,000 timesteps. This is still slower learning compared to the Default Reward and the previous results with GPT-4o-2024-08-06.

```
# Total reward
total_reward = reward_from_enemy_damage + reward_from_attack + time_penalty + win_reward
```

Figure 5.33: Reward components, iteration 1 experiment 1

The strategy used here is also quite basic. It relies on rewards for attacking, damage dealt to enemies, and a time penalty to encourage quicker episode completion. The win reward is a sparse reward, granting +100 for winning and -50 for losing.

For the second experiment, the first two iteration have some wins early but the learning is instable, the iteration 2 reward function is more stable but learn slowly. We will see what strategy is used.

```
# === 5. Enemy visibility reward ===
curr_enemy_in_sight = 0
for i in range(num_enemies):
    for obs in observation:
        idx = base_offset + i * obs_per_enemy
        if obs[idx] == 1:
            curr_enemy_in_sight += 1
            break  # count each enemy only once

prev_enemy_in_sight = reward_components.get("enemy_in_sight_prev", curr_enemy_in_sight)
reward_enemy_in_sight = (curr_enemy_in_sight - prev_enemy_in_sight) * 0.2
reward_components["enemy_in_sight_prev"] = curr_enemy_in_sight

# === Total Reward ===
total_reward = (
    reward_enemy_damage +
    reward_attack_used +
    reward_step_penalty +
    reward_victory +
    reward_defeat +
    reward_enemy_in_sight
)
```

Figure 5.34: Part of Code Experiment 2 Iteration 2

As before, the total reward is calculated with significant emphasis on attack actions, enemy damage, and victory. However, an additional component has been introduced: the reward for enemies in sight. This reward component encourages agents based on how many enemies they can see. It counts the number of enemies visible for each agents and compares it to the previous count of visible enemies. If the number of visible enemies has increased since the last step, the agent is rewarded with a positive value, scaled by 0.2. Encouraging agents to spot enemies helps prevent situations where they might flee from combat. The scale of the reward for this reward function is quite small, 20 for a win and -20 for a loss.

The last experiment failed; even though it attempted to implement the same strategy as before, it was unsuccessful.

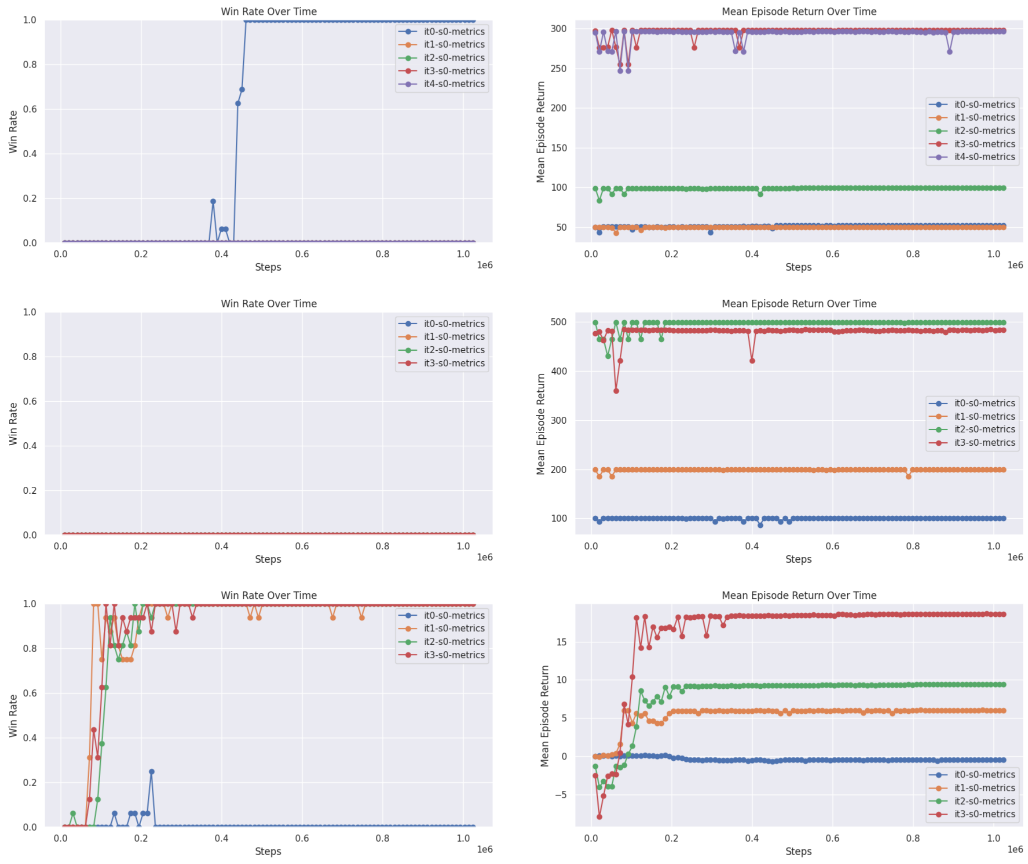


Figure 5.35: o3-mini Reward in a 4 marines VS 3 marines scenario with IPPO algo and no micro Strategies given

In the first experiment, only iteration 0 shows an increasing win rate. The second experiment fails, while the last experiment succeeds. In the last experiment, most iterations learn how to win quickly, around 50,000 timesteps. The reward components are the same as those used in gpt-4o and gpt-4-latest:

```
# ---- Total Reward Computation ----
total_reward = (
    enemy_health_reward_scaled +
    aggressive_bonus -
    time_penalty +
    win_bonus +
    loss_penalty -
    max_step_penalty
)
```

Figure 5.36: Reward Computation iteration 1 Experiment 3

Indeed, the reward components focus, as before, on enemy health, damage dealt to enemies, a sparse win bonus, and a time penalty.

In conclusion, the results show clear differences in how each model handles reward function design depending on the guidance given. GPT-4o-2024-08-06 previously struggled when instructed to implement the "Dancing" strategy; it focused heavily on that specific but complex behavior and failed to represent it effectively in the reward function. However, when no strategy was provided in the prompt, it reverted to basic approaches like rewarding enemy health reduction that worked well in the simpler scenario. GPT-4o-latest behaved similarly: although it previously managed to implement Focus Fire when prompted with Dancing prompt, we expected it to attempt more advanced strategies in this case. But, it also defaulted to simple but effective techniques, just like its earlier version. Finally, o3-mini showed the most interesting behavior. Without any strategy explicitly provided, it actually performed better than when we asked it to implement Focus Fire. This suggests that o3-mini benefits from less constrained prompts, allowing it to form simpler, more general reward functions that are well suited for straightforward tasks.

Test With More Recent Models

In this section, we evaluate more recent models GPT-4.1 and o4-mini which are expected to offer improved performance. However, this may not necessarily be the case, as earlier results showed that gpt-4o-latest did not outperform its fixed-version with the Focus Fire prompt. It's important to note that the following experiments were conducted using the same prompts as in the previous tests, the same of the 4 marines versus 3 marines scenario.

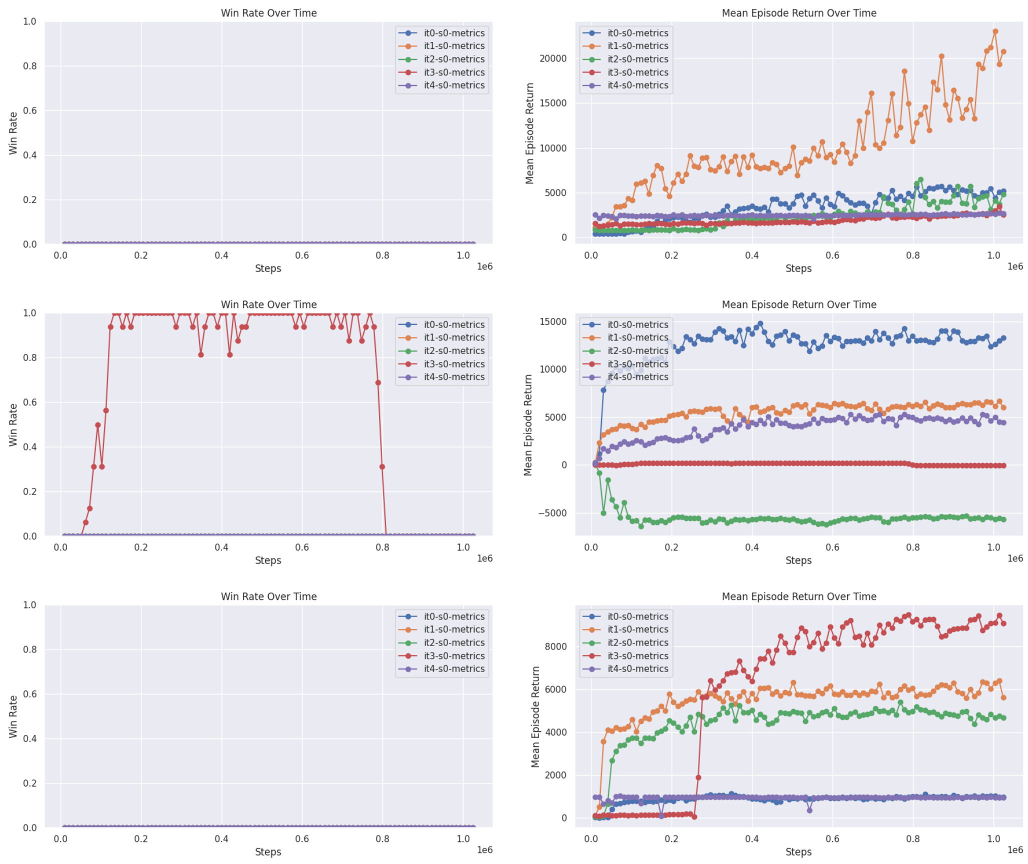


Figure 5.37: gpt-4.1 Reward in a 4 marines VS 3 marines scenario with IPPO algo and with Focus Fire

From the three experiments in the 4 vs 3 scenario using GPT-4.1, we can see that the results are not what we expected for a 'supposed' better model. The first and last experiments did not produce any well designed reward functions. The only interesting reward function comes from the third iteration of the second experiment. This one is unusual because it manages to reach around 100% win rate, but then suddenly drops after 800,000 timesteps. It's possible that the agent learned to hack the reward after 800,000 timesteps, which could explain this strange behavior during training. But the reward hacking is also unlikely since we can notice that the mean reward slightly drops around 800,000 timesteps, suggesting that the reward function reflects the sudden losses after winning. Possibly an anomaly during the Mava training.

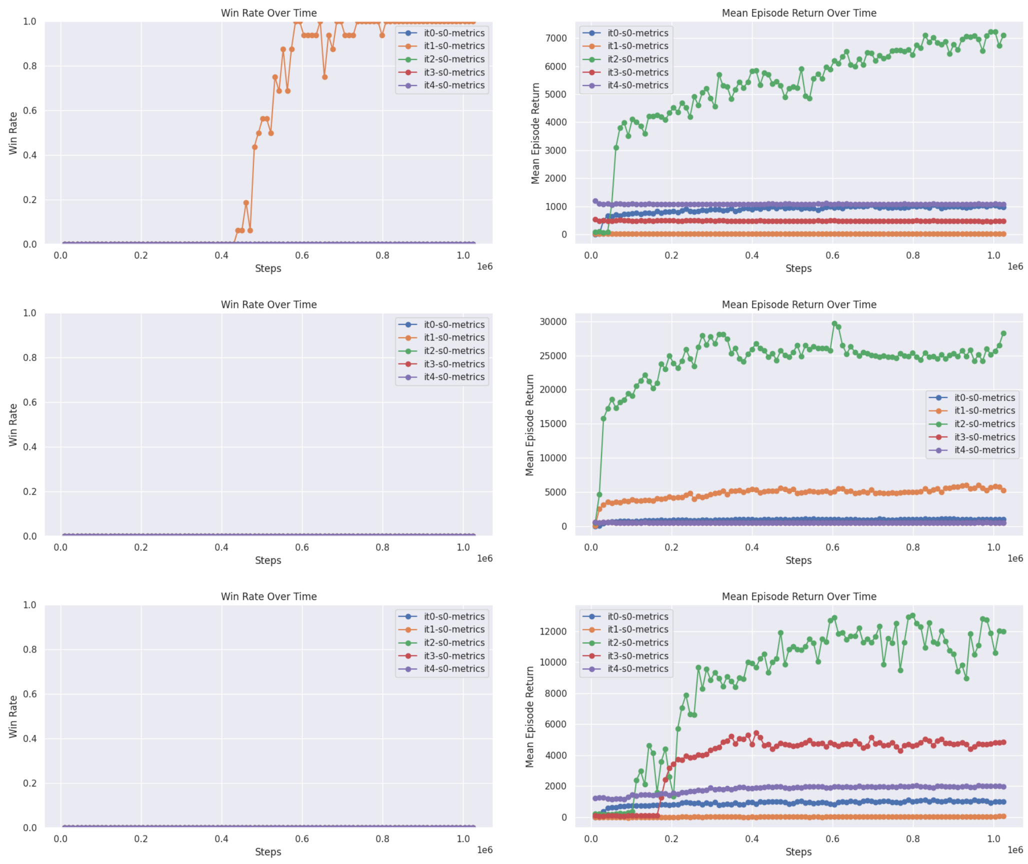


Figure 5.38: o4-mini Reward in a 4 marines VS 3 marines scenario with IPPO algo and Focus Fire prompt

From the graphs above, it is clear that neither GPT-4.1 and o4-mini perform well on this task. In the simpler 4 vs 3 scenario, GPT-4o significantly outperforms both, generating reward functions that match the performance of the human-designed baseline for this scenario. In contrast, GPT-4.1 fails to produce any effective reward function for the Focus Fire prompt. Interestingly, o4-mini does manage to generate one reward function that eventually reaches a 100% win rate, but it does so much more slowly after approximately 400,000 time steps whereas GPT-4o starts improving around the 50,000 time steps.

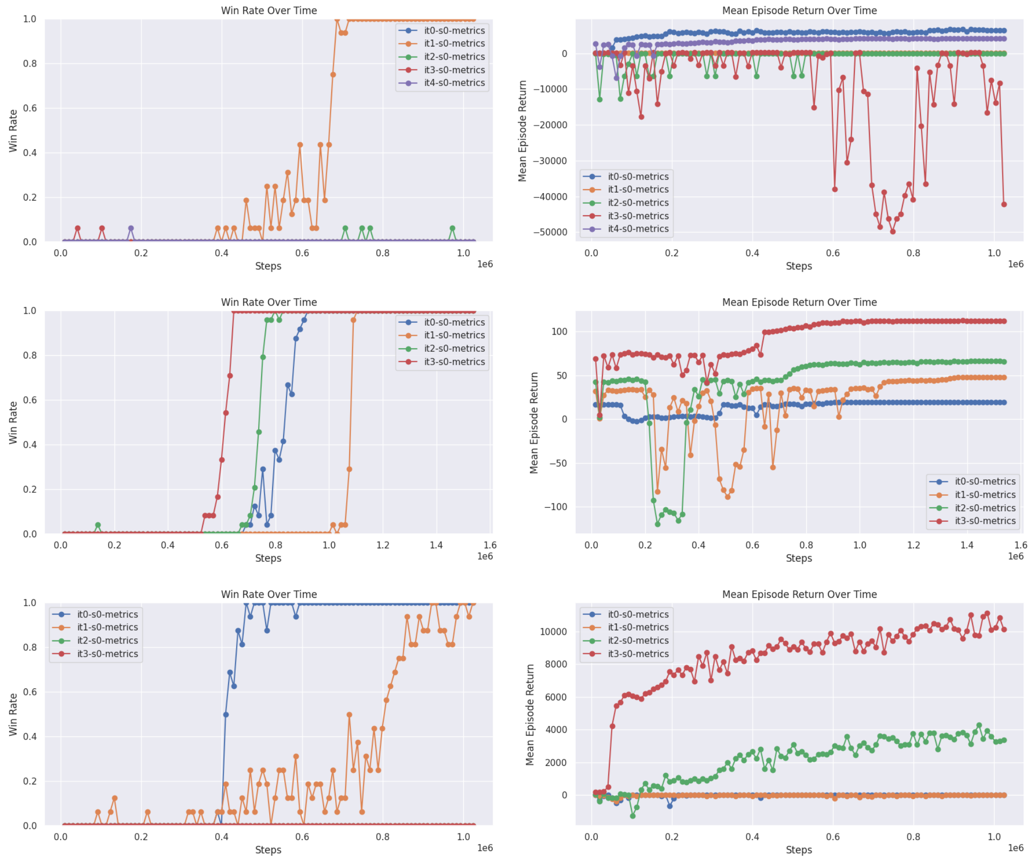


Figure 5.39: gpt-4.1 Reward in a 4 marines VS 3 marines scenario with IPPO algo and no micro Strategies given

Additionally, we ran an experiment where GPT-4.1 was prompted without explicit mention of micro-strategies. While the resulting reward function performed slightly better than with the Focus Fire prompt, it still lagged behind previous results. It generates one promising reward function, but learning progresses slowly, with the first wins only appearing after approximately 400,000 timesteps.

These findings highlight that newer models are not necessarily better for every task. While GPT-4.1 and o4-mini may benefit from larger context windows and improved reasoning in some domains, they appear to regress in this specific application.

5.3 Results Overview

This section summarizes how the three most-tested models; GPT-4o, GPT-4o-latest, and o3-mini, perform in each scenario. It shows the average overall Max win rate, calculated from every reward function created in all Eureka iterations and experiments. In other words, for a given experiment, it calculates the maximum win rate observed. Then, it repeats this for all instances of the same experiment and computes the average of these maximum win rates.

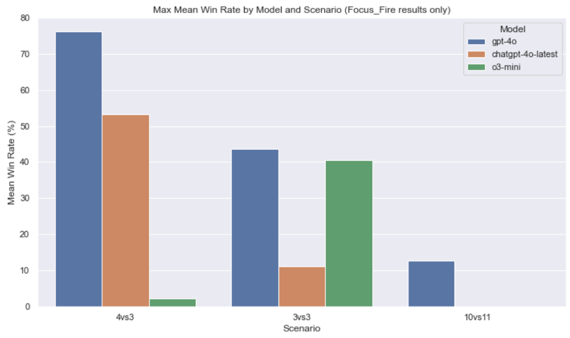


Figure 5.40: Focus Fire Mean win rate for each LLM models and Scenarios

The first graph shows how each model performs with the Focus Fire prompt. In the 4 vs 3 scenario, GPT-4o has a mean max win rate of 76.25%. GPT-4o-latest follows at 53.19%, while o3-mini is almost ineffective at 2.08%. When the scenario shifts to 3 vs 3, every model's Max win rate drops: GPT-4o falls to 43.75%, o3-mini unexpectedly climbs to 40.6%, and GPT-4o-latest sinks to 11%. GPT-4o still performs best overall. In the toughest 10-vs-11 scenario, GPT-4o has a mean max win rate of 12.7%, GPT-4o-latest and o3-mini records a 0% mean max win rate, highlighting how challenging this setup is for the models.

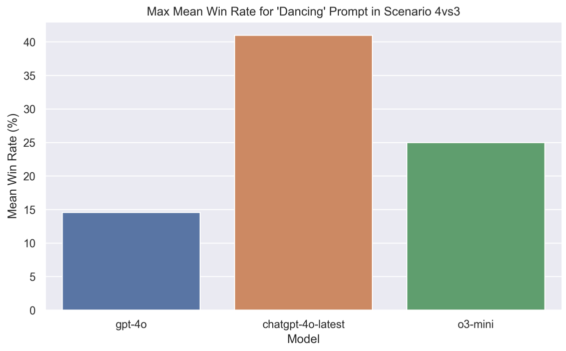


Figure 5.41: Dancing Mean Win rate for each LLM models in 4 vs 3

This graph shows the average max win rate results using the Dancing prompt in a 4v3 scenario. As we can see, GPT-4o did not succeed in this task, it mostly failed to produce any reward function that could train the agents effectively, with an average max win rate of only 14.58%. On the other hand, GPT-4o-latest performed better, reaching an average max win rate of 41%, suggesting that this newer version of the model takes more initiative on its own. O3-mini performs not badly in this experiment with a mean max win rate of 25%. As previously noted, it was the only model that better implemented the Dancing strategy.

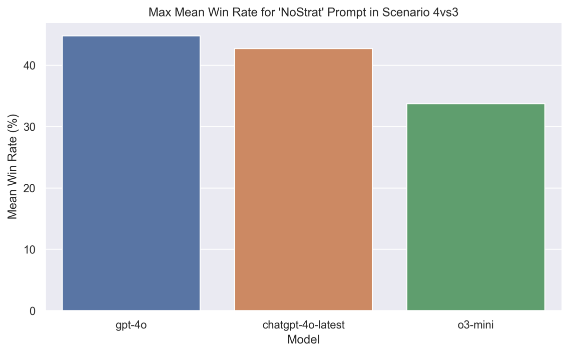


Figure 5.42: No Micro Strategies Mean Win rate for each LLM models in 4 vs 3

The graph above shows average win rates for the models in the 4vs3 scenario when the prompts include no hints about which micro-strategies to use. When each model must invent its own tactics, GPT-4o leads with a 44.8% win rate, GPT-4o-latest follows at 42.7%, and o3-mini trails at 33.75%.

Overall, the results show that GPT-4o performs best in straightforward scenarios, especially with clear strategies like Focus Fire or no strategy at all. It consistently generates effective reward functions early but struggles in complex environments like 10vs11. GPT-4o-latest shows some improvement in strategic initiative, performing better than GPT-4o and o3-mini in the Dancing scenario, but it still mostly relies on the same basic reward components and lacks consistent performance. O3-mini performs poorly in 4vs3 Focus Fire configuration but stands out in the Dancing prompt, being the only model to implement a reward function that truly reflects the strategy by considering enemy positions and agent retreat. It also generalizes reasonably well in simple tasks without any micro-strategy given in the prompts. In summary, GPT-4o is the most reliable overall, GPT-4o-latest is slightly more creative, and o3-mini shows potential in specific strategic contexts.

5.4 Deeper Comparison With the Default Reward

Now that we have completed and analyzed all our experiments, we can summarize our findings and draw conclusions to assess whether the LLM-generated reward functions are better than the Default Reward.

To begin with, let's recall that the Default Reward is quite simple. The goal in the SMAC environment is to defeat all enemy units. The Default Reward encourages this behavior by giving positive feedback for reducing enemy health, killing enemies, and ultimately winning an episode. So while the strategy is straightforward, the real challenge lies in designing a reward function that effectively encourages agents to adopt this behavior through training.

One key strength of the Default Reward is its generality it works across different SMAC scenarios (e.g., 4v3, 3v3, 10v11 with shields, and mixed unit types). On the other hand, the reward functions generated by LLMs are scenario-specific, as we tailor the prompts to the environment. This is useful, as it allows us to give precise examples of the observation and action spaces for the model to understand the task better.

From the experiments, especially in the simpler 4 vs 3 and 3 vs 3 scenarios, we observed that both GPT-4o and O3-mini were capable of producing reward functions that trained agents as quickly as, and in some cases faster than, the Default Reward. However, in the more complex 10 vs 11 scenario, none of the models succeeded in producing a reward function that performed as well. This is a major limitation of LLMs's reward functions identified in this thesis.

Understanding why this happens is crucial. One issue is that LLMs can sometimes generate invalid reward functions, by invalid we mean code that contains syntax errors (which did not occur in our tests), or with logical or indexing mistakes, meaning that the LLM didn't clearly understand the observation space. While such errors were rare in simple environments, they occurred more frequently in complex scenarios like 10 vs 11. This suggests that even when the prompt is well-structured, the increased complexity can lead the LLM to misinterpret or mishandle the data. It's also possible that the reward code executes without errors but still interprets the game state incorrectly, leading to ineffective rewards.

Interestingly, we also found that when no specific micro-strategy is requested, the LLMs tend to generate basic and effective strategies on their own, such as giving rewards for damaging enemies or performing attack actions, much like the Default Reward. This indicates that LLMs do understand the core components of a good reward function. However, the issue lies in the implementation details, where small mistakes can significantly affect performance.

There are several other factors that can influence performance, particularly in complex scenarios. First, reward scaling might not always be well-adjusted. This is supposed to be addressed through the iterative reflection process, but it may not always be enough. Finally, our ability to explore more iterations and samples is limited by computational constraints. Running more extensive experiments would be ideal, but was not feasible within the scope of this thesis.

In conclusion, while LLMs like GPT-4o and O3-mini can successfully generate reward functions for simple tasks, they still struggle with complex scenarios due to errors in observation handling, reward logic, and limited generalization. Their performance is promising, but not yet consistent or robust enough to replace manually designed, general-purpose reward functions like the Default Reward, especially in more difficult environments.

6 Limitations & Future Work

In this section, we discuss the potential limitations and possible future work of this master thesis.

6.1 Limitations

One major limitation of this master thesis is the long training time required for RL experiments. Limited computational resources significantly constrained the number of experiments we could perform. Access to more powerful hardware would have enabled broader testing. The experiments have been performed on 2 different machines:

Component	**First Machine**	**Second Machine**
CPU	i5-10210U	i5-12400F
RAM	8.00 GB	16.00 GB
GPU	No GPU	RTX 3060Ti
Storage	500 GB SSD	1 TB SSD

Table 6.1: Comparison of the two machines used for experiments

For the first machine, the experiments for the 4vs3 and 3vs3 scenarios took approximately 70 minutes, a bit over an hour, for a single sample of one iteration of the Eureka experiment. This time must be multiplied by the total number of iterations performed and the number of samples generated in each iteration. In general, one sample was produced per iteration, and four iterations were performed. Thus, an entire experiment generally took around 280 minutes (±4 hours and 30 minutes) on this machine, for a training of 1,000,000 time steps. For the 10vs11 scenario, we needed double the number of time steps, as it is a more challenging setup, but these experiments were carried out the other machine.

On the second machine, the 4vs3 and 3vs3 scenarios took about 30 minutes each, meaning an entire experiment lasted roughly 2 hours, which was quicker than on the first machine.

However, this machine was only available on weekends, as it was a fixed PC. The 10vs11 experiments, on the other hand, took two to three times longer than the 4vs3 and 3vs3 experiments, as they required twice the number of time steps and each episode lasted longer due to the increased number of units. This justifies the limitations introduced by the longer training times, the need to repeat the process for each LLM-generated reward function during the Eureka process, and the constraints imposed by the available hardware resources.

Training in RL and LLMs are inherently unstable processes. When combined, they introduce even greater instability. This work attempted to reduce the impact of this instability by conducting multiple runs of the same experiments, aiming for more consistent results. However, while the results provide a general sense of performance, they lack the consistency needed to draw definitive conclusions or generalizations. Therefore, the findings should be viewed as indicative rather than conclusive.

Additionally, considerable time was lost attempting to use the ePyMARL framework to train agents in SMAClite. Unfortunately, ePyMARL did not produce stable or consistent results in this context. As a result, we switched to the Mava framework, which proved to be a better fit for our task.

Another limitation is related to the use of LLMs for reward function design. While LLM-generated reward functions performed comparably to human-designed ones in simple scenarios, the time spent crafting effective prompts and setting up the Eureka reward design process may not yield a clear time-saving benefit. In some cases, designing a consistent reward function manually may be more straightforward and reliable.

Finally, our experiments were limited to relatively simple scenarios; basic 4vs3 marine engagements on flat maps without shields or unit diversity. This restricts the generalization of our findings to more complex or realistic environments. The reward generated by the LLM are specific to a scenario we can't use it for any kind of scenario.

6.2 Future Work

A natural extension of this work would be to evaluate the proposed LLM-guided reward generation framework in more complex SMACLite scenarios, incorporating elements such as multiple unit types, shielded units, and more intricate or asymmetric maps. These environments more closely resemble real-world tactical challenges and would provide a valuable test of the robustness and generalization capabilities of LLMs when tasked with reward design. As the tactical space becomes richer, the LLM must reason about more diverse interactions, resource trade-offs, and longer-term strategic goals, offering a fertile ground for studying its limitations and potential enhancements.

Another promising direction involves incorporating visual inputs such as frames or state snapshots from episode replays into the reward reflection pipeline. Rather than relying only on structured logs and numerical statistics, the LLM could be prompted with visual representations of agent behavior, such as situations where allies retreat from combat or flank enemies. This could enable the model to ground its reasoning in more human-like perception, potentially producing reward functions that better reflect intuitive notions of good behavior. While this direction was considered during the course of this thesis, practical constraints on time and tooling prevented its implementation. Future work could involve developing pipelines to encode visual observations into descriptive prompts or leveraging multi modal LLMs capable of processing both text and images.

As discussed earlier, a key limitation of the current framework is the scenario specific nature of the generated reward functions. This design decision was primarily motivated by the need for prompt clarity, as providing the LLM with detailed, scenario specific context simplified the reward interpretation task. However, this approach limits generalization and requires repeated manual setup for each new scenario. A natural line of improvement would be to explore the generation of generalized, reusable reward functions through the use of more abstract and high level prompts. Future efforts might combine prompt engineering, few-shot learning, or fine-tuning strategies to enhance the model's ability to reason across diverse environments.

Another compelling direction for future work is the integration of human-in-the-loop reward design, where domain experts or end users actively participate in refining or guiding the reward function generation process. While this thesis demonstrates the potential of LLMs to autonomously propose reward functions based on scenario descriptions and observations, such models may still generate objectives that are misaligned with human expectations and omit subtle tactical nuances. By incorporating human feedback either through preference selection or iterative prompt adjustments, the reward generation process could become more aligned with expert intuition and higher level goals. For example, a human observer could be asked to rate or comment on agent performance under a generated reward, and this feedback could be used to either update the LLM prompt or directly influence reward function structure. This interactive loop may lead to more interpretable and robust reward functions. Implementing such a human-in-the-loop system would require careful interface design, efficient feedback mechanisms, and evaluation metrics to measure improvement over purely autonomous approaches.

7 Conclusion

This thesis looked at how RL and LLMs can work together, focusing on how LLMs can help create reward functions that guide agent behavior in multi-agent tasks. By adapting the Eureka framework and using the SMACLite environment, we showed that LLMs can automatically create reward functions that help agents learn in some simple situations, but they still struggle with more difficult ones. The experiments showed that models like GPT-4o and o3-mini can generate useful reward functions, but their success depends a lot on how the prompt is written, how hard the environment is, and what strategies are used. Among the newer models tested, such as GPT-4.1 and O4-mini, the results did not clearly show better performance compared to older models, though they were only tested on a few simple scenarios. A major challenge remains the unstable and unpredictable nature of RL training, made worse by the technical limits of some frameworks like ePyMARL. In contrast, the Mava framework proved to be more reliable and was used for more consistent testing. This work also showed how important it is to carefully design prompts and choose the right model, as both can introduce unwanted bias. The main limits of this study were the simplicity of the scenarios and the fact that the reward functions were specific to each one, making it hard to apply them to other cases. Also, because of time limits, we could not run as many experiments or try as many models and setups as we wanted. With more time, we would have explored more scenarios, prompts, and training options. Still, this thesis lays the groundwork for future research on using LLMs to help design reward functions. Future work could include testing in more complex and realistic environments, using different types of input like images or sound, and involving humans in the loop. These paths could help make AI systems using RL and LLMs more flexible and effective.

Bibliography

References

[1] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[2] Fionn Murtagh. Multilayer perceptrons for classification and regression. *Neurocomputing*, 2(5-6):183–197, 1991.

[3] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[4] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[5] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[6] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2025.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *arXiv*, 2020.

[8] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothee Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *arXiv*, 2023.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[10] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 1996.

[11] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[12] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewald, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 2019.

[13] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE*, 2008.

[14] Chuanneng Sun and Dario Pompili. Llm-based multi-agent reinforcement learning: Current and future directions. *arXiv*, 2024.

[15] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[16] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of PPO in cooperative multi-agent games. In *Advances in Neural Information Processing Systems*, 2022.

[17] Christian Schroeder De Witt, Tarun Gupta, Denys Makoviichuk, Viktor Makoviychuk, Philip HS Torr, Mingfei Sun, and Shimon Whiteson. Is independent learning all you need in the starcraft multi-agent challenge? *arXiv preprint arXiv:2011.09533*, 2020.

[18] Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. Podracer architectures for scalable reinforcement learning. *arXiv preprint arXiv:2104.06272*, 2021.

[19] Yecheng Jason Ma, Guanzhi Wang William Liang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi “Jim” Fan, and Anima Anandkuma. Eureka: Human-level reward design via coding large language models. *arXiv*, 2024.

[20] Adam Paszke, , Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *33rd Conference on Neural Information Processing Systems*, 2019.

[21] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philiph H. S. Torr, Jakob Foerster, and Shimon Whiteson. The StarCraft Multi-Agent Challenge. *CoRR*, abs/1902.04043, 2019.

[22] Georgios Papoudakis, Filippos Christianos, Lukas Schäfer, and Stefano V. Albrecht. Benchmarking multi-agent deep reinforcement learning algorithms in cooperative tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks (NeurIPS)*, 2021.

[23] Ruan de Kock, Omayma Mahjoub, Sasha Abramowitz, Wiem Khlifi, Callum Rhys Tilbury, Claude Formanek, Andries P. Smit, and Arnu Pretorius. Mava: a research library for distributed multi-agent reinforcement learning in jax. *arXiv preprint arXiv:2107.01460*, 2023.

[24] Adam Michalski, Filippos Christianos, and Stefano V. Albrecht. Smaclite: A lightweight environment for multi-agent reinforcement learning. 2023.

[25] Thomas F. Heston and Charya Khun. Prompt engineering in medical education. *International Medical Education*, 2(3):198–205, 2023.

[26] Xuechunzi Bai, Angelina Wang, Ilia Sucholutsky, and Thomas L Griffiths. Measuring implicit bias in explicitly unbiased large language models. *arXiv preprint arXiv:2402.04105*, 2024.

[27] Ilya Kostrikov, Kumar Krishna Agrawal, Debidatta Dwibedi, Sergey Levine, and Jonathan Tompson. Discriminator-actor-critic: Addressing sample inefficiency and reward bias in adversarial imitation learning. *arXiv preprint arXiv:1809.02925*, 2018.

Appendices

.1 Prompts

```
You are a reward engineer trying to write reward functions to solve reinforcement
  learning tasks as effective as possible.
Your goal is to write a reward function for the environment that will help the agent
  learn the task described in text.
Your reward function should use useful variables from the environment as inputs.
The reward function signature must be:
{task_reward_signature_string}
Don't use UTF emojis/Colored Symbols in your responses and also the only code that we
  will be used, is the one in the compute_step_reward function, don't write code
  outside of this function otherwise it might cause problems.
```

Figure 1: Initial_system.txt

```
The Python environment description is:
{task_obs_code_string}

IMPORTANT: The Max number of steps per episode for this Environment is {time_limit},
  once reached, the episode ends.

Write a reward function for the following task: {task_description}
Ensure to use the compute_step_reward function to solve this task. It is of the
  highest priority and must be completed accurately and promptly.
```

Figure 2: Initial_user.txt

```
We trained a reinforcement learning (RL) policy using the provided reward function
  and tracked the values of the individual components
in the reward function during training for a number of episodes equal to the length
  of the features provided for each reward component.
Specifically, we tracked the episode_lengths and success_rate, where the success rate
  is defined as 1 if the episode ends with all enemies
defeated and 0 otherwise. Additionally, we tracked the cumulative sum of reward
  components across each episode.
For each cumulated sum of tracked value, we computed the following statistics: the
  Mean, Standard Deviation, Max and Min;
Here are the values encountered:
```

Figure 3: policy_feedback.txt

```
Please carefully analyze the policy feedback and provide a new, improved reward
  function that can better solve the task. Some helpful tips for analyzing the policy
  feedback:
    (1) If the success rates are always near zero, then you must rewrite the entire
      reward function, zero success rate all the time is either because all allies
      died or either if the model is to slow to kill enemies, take that into account
      in your refinement
    (2) If the values for a certain reward component are near identical throughout,
      then it may indicate that the component is already at its optimized value or
      that RL is not able to optimize this component as it is written. You may
      consider
        (a) Changing its scale or the value of its temperature parameter
        (b) Re-writing the reward component
        (c) Discarding the reward component
    (3) If some reward components' magnitude is significantly larger, then you must
      re-scale its value to a proper range
    (4) For the spare reward, make sure to adapt the attribution conditions if
  necessary, accordingly to its values.


You Should use the Focus Fire strategy: the agent should prioritize a focus fire
  strategy, where multiple units coordinate their attacks on a single enemy at a
  time. This  approach maximizes damage output, eliminates threats faster, and
  reduces incoming damage by quickly lowering the number of active enemies. The
  reward function should encourage this behavior by providing higher rewards when
  units collectively target and eliminate an enemy rather than distributing attacks
  across multiple enemies.
Please analyze each existing reward component in the suggested manner above first,
  and then write the reward function code.
Please make sure every component uses a substracting formula with the previous one as
  follow:
    # Exemple usage of the reward_components : conservative formula, reward for
  lowering the criterion
    criterion = torch.abs(observation[0])  # Get the absolute value of the current
  criterion from the observation
    previous_criterion = reward_components.get('criterion', criterion)  # Retrieve
  the previous criterion, default to current criterion if not available
    component_reward = previous_criterion - criterion  # Reward is higher if
  criterion decreases
```

Figure 4: code_feedback.txt

```
The output of the reward function should consist of two items:
    (1) the total reward (NDArray),
    (2) a dictionary of each individual reward component.
The code output should be formatted as a python code string: "```python ... ```".

Some Important helpful tips for writing the reward function code (you should follow
  them):
    (1) You can enhance the task by incorporating sparse rewards for successes and
      failures. Consider key events or milestones in the environment that would
      benefit from such reward adjustments.
    (2) Most importantly, you should subtract the current evolution of the components
      from the previous one to track their progress towards the goal.
        - conservative : previous-current
        - not conservative : -current or exp(-current)
    (3) The reward code's input variables must contain only attributes of the
      provided environment class definition (namely, variables that have prefix
      self.). Under no circumstance can you introduce new input variables.
    (4) You should use negative reward when the episode last for too long. The
  maximum number of steps is given in the environment description, after this is
  reached, the episode ends.
    (5) Use this Strategy: To improve combat efficiency in SMACLite, the agent should
  prioritize a focus fire strategy, where multiple units coordinate their attacks on
  a single enemy at a time. This  approach maximizes damage output, eliminates
  threats faster, and reduces incoming damage by quickly lowering the number of
  active enemies. The reward function should encourage this behavior by providing
  higher rewards when units collectively target and eliminate an enemy rather than
  distributing attacks across multiple enemies.
    (6) Looking at the actions made by the agents trained is a good practice
```

Figure 5: code_output_tip.txt

~~~
```python
def compute_step_reward(
    observation: Tuple[np.ndarray, ...],
    action: List[int],
    done: bool,
    reward_components: Dict[str, float],
    step: int
) -> Tuple[np.ndarray, Dict[str, float]]:
    """
    Compute the reward for a step in an environment.

    Parameters:
        observation (Tuple[np.ndarray, ...]): The current observation of the
  environment.
        action (List[int]): The action taken by the agent.
        done (bool): A flag indicating if the episode is done.
~~~

```
        reward_components (Dict[str, float]): A dictionary containing the reward
  components from the previous step.
        step (int): The number of the step.

    Returns:
        Tuple[np.ndarray, Dict[str, float]]:
            - The total computed reward as an NDArray repeated for each agent.
            - Updated dictionary of reward components for the current step.
    """

    # Return total reward as a NumPy array repeated for each agent
    return np.full(num_agents, total_reward, dtype=np.float32), reward_components
```
```

Figure 6: reward_signature.txt (mava)

````
```python
def compute_step_reward(observation, action, done, reward_components,step):
    """
    Compute the reward for a step in an environment.

    Parameters:
        observation (List[torch.Tensor]): The current observation of the environment.
        action (numpy.ndarray): The action taken by the agent.
        done (bool): A flag indicating if the episode is done.
        reward_components (Dict[str, torch.Tensor]): A dictionary containing the
  reward components from the previous step.
        step (int): the number of the step.

    Returns:
        total_reward (float): The total computed reward for the current step.
        reward_components (Dict[str, torch.Tensor]): Updated dictionary of reward
  components for the current step.
    """

    # Exemple usage of the reward_components : conservative formula, reward for
      lowering the criterion
    criterion = torch.abs(observation[0])  # Get the absolute value of the current
      criterion from the observation
    previous_criterion = reward_components.get('criterion', criterion)  # Retrieve
      the previous criterion, default to current criterion if not available
    component_reward = previous_criterion - criterion  # Reward is higher if
      criterion decreases

    ...

    return total_reward, reward_components
```
~~~

Figure 7: reward_signature.txt (ePyMARL)

## .1.1 Environment Description Prompts

```
## Environment Description (3v3)

The `SMACliteEnv` is a multi-agent battle environment where 3 allied agents face off
  against 3 enemy units. Each agent operates independently with limited vision (sight
  range). The objective is to eliminate all enemies as quickly and aggressively as
  possible.

Strategy Rule: Go all in. Be hyper-aggressive. Do not prioritize ally health. Strike
  fast and hard to prevent the enemy from reacting.

---
```
~~~

## Observation Structure (Per Agent)

Total Length: 30 values
(= 4 for movement + 15 for enemies + 10 for allies + 1 for own health)

---

### 1. Movement Capabilities
- Size: 4
- Format: `[move_N, move_S, move_E, move_W]`
- Example: `[1, 0, 1, 1]`

---

### 2. Enemy Unit Info (3 enemies x 5)
- Size: 15
- Format per enemy: `[attack_available, distance, dx, dy, health]`
- Example for one enemy: `[1, 0.5, 0.1, -0.3, 0.8]`
- If not visible or dead: `[0, 0, 0, 0, 0]`

---

### 3. Ally Unit Info (2 allies x 5)
- Size: 10
- Format per ally: `[alive, distance, dx, dy, health]`
- Example: `[1, 0.3, 0.1, -0.2, 0.85]`
- If not visible or dead: `[0, 0, 0, 0, 0]`

---

### 4. Own Features
- Size: 1
- Format: `[health]`
- Example: `[0.75]`

---

## Observation Example Format

```python
[
  1, 0, 1, 1,  # Movement

  # Enemy 0
  1, 0.5, 0.1, -0.3, 0.8,
  # Enemy 1
  1, 0.7, -0.2, 0.1, 0.6,
  # Enemy 2
  0, 0, 0, 0, 0,

  # Ally 0
  1, 0.3, 0.5, -0.2, 0.9,
  # Ally 1
  1, 0.2, -0.1, 0.4, 1.0,

  0.75  # Own health
]

Full Observation (All Agents)
If there are 3 agents, the full observation is a tuple of 3 arrays:

(
  obs_agent_0,
  obs_agent_1,
  obs_agent_2
)
Each obs_agent_i is a NumPy array of length 30.
```

```


## Action Space (Per Agent)
Each agent selects one discrete action at every timestep.
The total number of actions is:

Action Space Size = 6 + number_of_enemies - 1 = 6 + 3 - 1 = 8

Action Index => Description
0    => NO-OP (Only for dead units)
1    => STOP (Do nothing)
2    => MOVE_NORTH
3    => MOVE_SOUTH
4    => MOVE_EAST
5    => MOVE_WEST
6    => ATTACK enemy 0
7    => ATTACK enemy 1
8    => ATTACK enemy 2

Note: ATTACK actions (6-8) are only available if:

- The target enemy is alive

- The target is within range


## Termination Conditions

An episode ends when any of the following occurs:

1.All Enemy Units are Eliminated
   - Episode ends in success and reward is given.

2.All Allied Agents are Eliminated
   - Episode ends in failure and penalty is applied.

3.Maximum Number of Steps Reached
   - Episode ends automatically due to time limit.
```

Figure 8: 3vs3_environment_description.txt

```

The SMACliteEnv is a multi-agent environment where agents are units in a battle scenario, and the goal is to strategically control and manage their actions to defeat enemy units while maintaining their own survival. The environment involves allied agents fighting against enemy units, with a focus on interactions such as movement, attacking, and possibly healing. The environment is partially observable, meaning each agent can only observe certain aspects of the world, particularly those within its sight range.

The objective is to maximize team performance by strategically commanding units to attack enemies. Your only goal is to defeat enemies, be very aggressive, don't let the enemies the time to react. Avoid giving too much importance to the health of allies. A typical episode ends when all enemies are defeated, all allied agents are eliminated, or the time limit for an episode is reached.

---

### Observation Structure : list(torch.tensor)

1. Movement Capabilities
- Size: `4 values`
- Description: Indicates whether the agent can move in the four cardinal directions (up, down, left, right).
- Example: `[1, 0, 1, 1]` (can move up, cannot move down, can move left and right).

2. Enemy Unit Information
```

```
- Size: `5` (per enemy)
- Description:
  - Attack Availability: `1` if the agent can attack, `0` otherwise (if 0, all fields
    are zero).
  - Distance: Normalized distance to the enemy.
  - Relative Position (dx, dy): X and Y position of the enemy relative to the agent.
  - Health: Enemy's current health (normalized).

3. Ally Unit Information
- Size: `5` (per ally)
- Description:
  - Alive Status: `1` if the ally is alive.
  - Distance: Normalized distance to the ally.
  - Relative Position (dx, dy): Relative position of the ally.
  - Health: Ally's current health (normalized).

4. Agent's Own Features
- Size: `1`
- Description: Agent's own health, as a fraction of max health.

---

## Example Observation

Scenario: 3 enemies and 3 allies in sight.

Observation vector:

[1, 0, 1, 1,                        # Movement
 1, 0.5, 0.1, -0.3, 0.8,            # Enemy 1
 1, 0.7, -0.2, 0.1, 0.6,            # Enemy 2
 0, 0, 0, 0, 0,                     # Enemy 3 (dead or out of range)
 1, 0.3, 0.5, -0.2, 0.9,            # Ally 1
 1, 0.2, -0.1, 0.4, 1.0,            # Ally 2
 0, 0, 0, 0, 0,                     # Ally 3 (dead or out of range)
 0.75]                              # Own Health

This array has a fixed size of 35 in our 4v3 scenario.

---

### Full Observation Tuple

If there are `n` agents in the environment, the full observation is:

(agent_obs_1, agent_obs_2, ..., agent_obs_n)

In our case (4 agents), this is a tuple of 4 numpy arrays.

---

## Action Space

Each agent selects one discrete action per step.

Action space size = 6 + number_of_enemies - 1
= 6 + 3 - 1 = 8
Valid actions: `0` through `8`

Available Actions:

0 => NO-OP (Only for dead units)
1 => STOP
2 => MOVE_NORTH
3 => MOVE_SOUTH
4 => MOVE_EAST
5 => MOVE_WEST
6 => ATTACK enemy 0
```

```
7 => ATTACK enemy 1
8 => ATTACK enemy 2

Conditions for ATTACK actions:
- Target must be alive
- Target must be within range

---

## Action Example

actions = [3, 7, 1, 5]

Interpretation:
Agent 1: MOVE_SOUTH
Agent 2: ATTACK enemy 1
Agent 3: STOP
Agent 4: MOVE_WEST

---

## Termination Conditions

An episode ends if:
1. All Enemies are Defeated -> Reward
2. All Allied Units are Defeated -> Penalty
3. Maximum Number of Steps Reached
```

Figure 9: 4vs3_environment_description.txt

```
## Environment Description (10v11)

The `SMACliteEnv` is a multi-agent battle environment where 10 allied agents face off
  against 11 enemy units. Each agent is controlled independently and can only observe
  within its sight range. The goal is to eliminate all enemies as aggressively as
  possible.

Strategy Rule: Be hyper-aggressive. Don't hesitate. Don't worry about ally health.
  Push the attack and overwhelm the enemy fast before they can react.

---

## Observation Structure (Per Agent)

Total Length: 105 values
(= 4 for movement + 55 for enemies + 45 for allies + 1 for own health)

---

### 1. Movement Capabilities
- Size: 4
- Format: `[MOVE_NORTH, MOVE_SOUTH, MOVE_EAST, MOVE_WEST]`
- Example: `[1, 1, 0, 1]`

---

### 2. Enemy Unit Info (11 enemies x 5)
- Size: 55
- Format per enemy: `[attack_available, distance, dx, dy, health]`
- Example for one enemy: `[1, 0.4, 0.2, -0.1, 0.9]`
- If not visible or dead: `[0, 0, 0, 0, 0]`

---

### 3. Ally Unit Info (9 allies x 5)
- Size: 45
- Format per ally: `[alive, distance, dx, dy, health]`
- Example: `[1, 0.3, 0.1, -0.2, 0.85]`
- If not visible or dead: `[0, 0, 0, 0, 0]`

---

### 4. Own Features
- Size: 1
- Format: `[health]`
- Example: `[0.75]`

---

## Observation Example Format

[
  # Movement
  1, 1, 0, 1,

  # Enemy 0
  1, 0.4, 0.2, -0.1, 0.9,
  # Enemy 1
  1, 0.6, -0.3, 0.2, 0.7,
  # Enemy 2
  1, 0.7, 0.0, -0.2, 0.65,
  # Enemy 3
  1, 0.5, 0.1, 0.3, 0.8,
  # Enemy 4
  1, 0.3, -0.2, 0.0, 0.6,
  # Enemy 5
  1, 0.9, 0.4, -0.4, 0.55,
  # Enemy 6
  0, 0, 0, 0, 0, # might be dead or out of range
  # Enemy 7
  0, 0, 0, 0, 0,
  # Enemy 8
  0, 0, 0, 0, 0,
  # Enemy 9
  0, 0, 0, 0, 0,
  # Enemy 10
  0, 0, 0, 0, 0,

  # Ally 0
  1, 0.3, 0.1, -0.2, 0.85,
  # Ally 1
  1, 0.4, -0.1, 0.3, 0.9,
  # Ally 2
  1, 0.6, 0.0, 0.2, 0.95,
  # Ally 3
  1, 0.7, 0.2, 0.0, 1.0,
  # Ally 4
  1, 0.2, -0.3, -0.1, 0.88,
  # Ally 5
  0, 0, 0, 0, 0, # might be dead or out of range
  # Ally 6
  0, 0, 0, 0, 0,
  # Ally 7
  0, 0, 0, 0, 0,
  # Ally 8
  0, 0, 0, 0, 0,

  # Own health
  0.75
]


Full Observation (All Agents)
If there are 10 agents, the observation is a tuple of 10 arrays:

```

```
(
  obs_agent_0,
  obs_agent_1,
  nos_agent_2,
  nos_agent_3,
  nos_agent_4,
  nos_agent_5,
  nos_agent_6,
  nos_agent_7,
  nos_agent_8,
  obs_agent_9
)
Each obs_agent_i is a numpy array of length 105.

## Action Space (Per Agent)

Each agent selects one discrete action at every timestep.
The total number of actions is:

Action Space Size = 6 + number_of_enemies - 1 = 6 + 11 - 1 = 16

### Action Index Mapping:

 Action Index => Description
0    => NO-OP (Only for dead units)
1    => STOP (Do nothing)
2    => MOVE_NORTH
3    => MOVE_SOUTH
4    => MOVE_EAST
5    => MOVE_WEST
6    => ATTACK enemy 0
7    => ATTACK enemy 1
8    => ATTACK enemy 2
9    => ATTACK enemy 3
10   => ATTACK enemy 4
11   => ATTACK enemy 5
12   => ATTACK enemy 6
13   => ATTACK enemy 7
14   => ATTACK enemy 8
15   => ATTACK enemy 9
16   => ATTACK enemy 10
Note: ATTACK actions (6-16) are only available if:
- The target enemy is alive
- The target is within range

---

## Termination Conditions

An episode ends when any of the following occurs:

1.All Enemy Units are Eliminated
   - Episode ends in success and reward is given.

2.All Allied Agents are Eliminated
   - Episode ends in failure and penalty is applied.

3.Maximum Number of Steps Reached
   - Episode ends automatically due to time limit.
```

Figure 10: 10vs11_environment_description.txt

## .2 Project Structure

### Main Files

```
|-- ePyMARL/
|-- mava/
|-- prompts/
|   |-- smaclite/
|   |   |-- code_feedback.txt
|   |   |-- execution_error_feedback.txt
|   |   |-- initial_system.txt
|   |   |-- policy_feedback.txt
|   |   |-- reward_signature.txt
|-- Early_stopping.py
|-- epymarl_mappo.py
|-- eureka.py
|-- experiment_config.yaml
|-- llm.py
|-- mava_ippo.py
|-- plot.py
|-- requirements.txt
|-- README.md
```

Figure 11: Mains Files from the Root directory structure.

### ePyMARL Experiments

```
|-- Smaclite_ePyMARL/
|   |-- Experiments/
|   |   |-- example/
|   |   |   |-- it0-s0-render/
|   |   |   |   |-- 2025-01-08_21-23-20/
|   |   |   |   |   |-- Smaclite-episode-0.mp4
|   |   |   |-- it1-s0-render/
|   |   |   |   |-- 2025-01-08_21-43-45/
|   |   |   |   |   |-- Smaclite-episode-0.mp4
|   |   |   |-- metrics/
|   |   |   |   |-- it0-s0/
|   |   |   |   |   |-- congif.json
|   |   |   |   |   |-- cout.txt
|   |   |   |   |   |-- info.json
|   |   |   |   |   |-- metrics.json
|   |   |   |   |   |-- run.json
|   |   |   |   |-- ...
|   |   |   |-- models/
|   |   |   |   |-- mappo_seed185390923_0_0/
|   |   |   |   |-- mappo_seed88704524_1_0/
|   |   |   |-- tmp/
|   |   |   |   |-- reward_components_tmp.json
|   |   |   |-- chat.txt
|   |   |   |-- config.json
|   |   |   |-- it0-s0-response.txt
|   |   |   |-- it0-s0-rewards_components.json
|   |   |   |-- plot.pdf
|   |-- custom_maps/
|   |   |-- 3m.json
|   |-- descriptions/
|   |   |-- v1/
|   |   |   |-- env_desc.txt
|   |   |   |-- task_desc.txt
|   |   |-- v2_Aggressive/
|   |   |   |-- env_desc.txt
|   |   |   |-- task_desc.txt
|   |-- rl_only/
|   |-- compute_reward.py
|    |-- env_utils.py
```

Figure 12: SMAClite ePyMARL files structures.

## Mava Experiments

```
|-- Smaclite_Mava/
|    |-- Experiments/
|    |    |-- example/
|    |    |    |-- json/
|    |    |    |    |-- it0-s0-metrics/
|    |    |    |    |    |-- marl_eval_plot.png
|    |    |    |    |    |-- metrics.json
|    |    |    |    |-- ...
|    |    |    |-- tmp/
|    |    |    |    |-- reward_components_tmp.json
|    |    |    |-- chat.txt
|    |    |    |-- config.json
|    |    |    |-- it0-s0-response.txt
|    |    |    |-- it0-s0-rewards_components.json
|    |    |    |-- mean_episode_return_plot.png
|    |    |    |-- win_rate_plot.png
|    |-- custom_maps/
|    |    |-- 3mvs3m.json
|    |    |-- 4mvs3m.json
|    |-- descriptions/
|    |    |-- v1/
|    |    |    |-- env_desc.txt
|    |    |    |-- task_desc.txt
|    |    |-- v2_Aggressive/
|    |    |    |-- env_desc.txt
|    |    |    |-- task_desc.txt
|    |    |-- ...
|    |-- rl_only/
|    |-- compute_reward.py
|    |-- env_utils.py
```

Figure 13: SMAClite Mava files structures.

The tree above illustrates the structure of the project, showcasing all the necessary folders and files required to run an experiment. For clarity and simplicity, a fictive experiment is illustrated; however, in practice, there are additional experiment folders with specific names, each corresponding to different experiment setups and configurations. Also for clarity and simplicity the ePyMARL and Mava folder have not been expanded, for more information about the structure of ePyMARL framework click here and Mava click here.

## .3 Use of AI

An AI tool has been used as a spell checker to catch any remaining errors after writing.