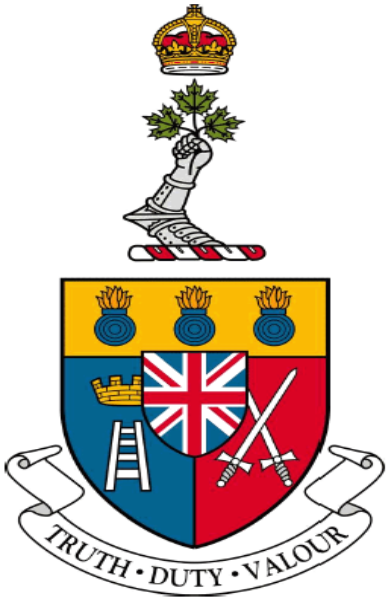
LARGE LANGUAGE MODEL INTEGRATION WITH REINFORCEMENT LEARNING TO AUGMENT DECISION-MAKING IN AUTONOMOUS CYBER OPERATIONS



A Thesis Submitted to the
Department of Electrical and Computer Engineering
by
Konur Tholl
Supervisor: Dr. Ranwa Al Mallah
Co-supervisor: Dr. Mariam El Mezouar

In Partial Fulfillment of the Requirements for the Degree of
Master of Applied Science in Computer Engineering

June, 2025

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Dr. Ranwa Al Mallah and Dr. Mariam El Mezouar, for their unwavering support, continuous guidance, and insightful feedback throughout this entire process. I am also deeply thankful to Dr. François Rivest for generously offering his expertise and time at a critical juncture.

To my parents, my brother, and the roomies - thank you for the constant motivation that helped carry me through. Finally, I want to give a shoutout to the guy who broke my finger in Jiu Jitsu, forcing me to focus on writing.

Abstract

Previous work has demonstrated the significant benefits of using Reinforcement Learning (RL) to solve complex problems in the cybersecurity domain. In particular, RL has great potential for real-world applications where agents can learn by directly interacting with an environment. This approach eliminates the need to prepare large datasets to train a model; however, it requires a suitable environment in which an agent can learn. The environment must simulate realistic cybersecurity conditions and provide the appropriate signals for an agent to learn an optimal policy.

Another significant advance in Artificial Intelligence (AI) is the development of Large Language Models (LLMs). An LLM's unique ability to recognize patterns in language makes it an invaluable asset for Autonomous Cyber Operations (ACO), where the goal is to incorporate autonomous decision-making in the cyber domain. Integrating an LLM provides simulated human reasoning in the RL process, allowing trained professionals to apply their invaluable skills elsewhere, greatly enhancing the scalability of cybersecurity.

Currently, the work done in ACO consists of RL agents that must begin learning from scratch in order to converge on a policy. Moreover, the knowledge of these agents is limited to the specific rules of the environment they are trained in. The purpose of this thesis is to introduce external knowledge in the RL pipeline to enhance decision-making and optimize training. This external knowledge will be in the form of an LLM that has already been trained in the cybersecurity domain.

Index Terms - Autonomous Cyber Operations; Large Language Model; Reinforcement Learning; Cybersecurity; LLM-RL Integration.

Résumé

Des travaux antérieurs ont démontré les avantages de l'utilisation de l'apprentissage par renforcement pour résoudre des problèmes complexes dans le domaine de la cybersécurité. En particulier, l'apprentissage par renforcement présente un grand potentiel pour les applications réelles où les agents peuvent apprendre en interagissant directement avec un environnement. Cette approche élimine le besoin de créer de grands jeux de données pour entraîner un modèle; cependant, elle nécessite un environnement dans lequel l'agent peut apprendre. Cet environnement doit simuler des conditions de cybersécurité réalistes et générer les signaux appropriés pour qu'un agent puisse apprendre une stratégie optimale.

Un autre avancée majeure en intelligence artificielle est le développement des grands modèles de langage. La capacité unique de ces modèles à comprendre le langage humain les rend pertinents pour les opérations cybernétiques autonomes où l'objectif est de considérer la prise de décision autonome dans le domaine cybernétique. L'intégration d'un grand modèle de langage offre une simulation du raisonnement humain dans le processus d'apprentissage par renforcement, permettant ainsi aux professionnels qualifiés d'appliquer leurs compétences précieuses ailleurs, ce qui améliore grandement l'évolutivité de la cybersécurité.

Actuellement, la recherche effectuée en opérations cybernétiques autonomes consiste en des agents d'apprentissage par renforcement qui doivent commencer à apprendre à partir de zéro afin de converger vers une stratégie optimale. De plus, la connaissance de ces agents est limitée aux règles spécifiques de l'environnement dans lequel ils sont entraînés. L'objectif de cette thèse est d'introduire des connaissances externes dans le processus d'apprentissage par renforcement pour améliorer la prise de décision et optimiser l'entraînement. Ces connaissances externes prendront la forme d'un grand modèle de langage déjà entraîné dans le domaine de la cybersécurité.

Mots-clés – Opérations cybernétiques autonomes; Grand modèle de langage; Apprentissage par renforcement; Cybersécurité; Intégration AML-GML.

Contents

List of Tables

List of Figures

1 Introduction

The extent to which offensive cyber operations have increased in recent years is substantial. To counteract this, cybersecurity must be able to scale at a proportional rate in a sustainable manner. Due to the sheer volume of data involved in cybersecurity, manually defending systems is not feasible. To address this concern, tools such as Intrusion Detection Systems (IDS) [1] have been created to assist analysts in detecting malicious activity; however, this still requires individuals to manually inspect each alert and act accordingly. Ideally, these tools would be capable of autonomously making decisions and executing actions, which would directly support cyber operators and greatly increase the scalability of cybersecurity. This is the primary goal of Autonomous Cyber Operations (ACO) [2]. Current applications of ACO are made possible through Reinforcement Learning (RL), a branch of Machine Learning (ML). However, traditional RL has its limitations including the following:

- The RL agent must learn from scratch for each task, increasing training time; and
- The agent's knowledge is scoped to the environment in which it is trained, limiting transferability.

These limitations can be mitigated by incorporating external knowledge into the RL process, which the agent can leverage to make decisions.

This study explores how a Large Language Model (LLM) can be integrated into the RL pipeline to enhance the agent's decision-making capabilities. This will reduce the training time required for an RL agent to converge onto an optimal policy in ACO.

1.1 Motivation

The sophistication and volume of offensive cyber operations have increased immensely over the past years. A Forbes study shows that the number of data breaches increased by 72% between 2021 and 2023 [3]. Cyber operators are no

longer able to manually defend their systems from these vast and increasingly complex attacks. It is imperative that tools are implemented to help defend these systems.

One of the first approaches to defend against these threats involves storing signatures of known attacks in a database [4]. Any new samples are compared against these stored signatures to determine if they are malicious. However, this reactive approach means that systems will be completely vulnerable to any new attack whose signature is not already in the database. This makes it very easy for adversaries to bypass defense systems by making minor modifications to their attacks (e.g., adding non-executable code to malware).

ML was introduced in cybersecurity to address these shortcomings due to its data-centric nature where it is able to recognize anomalies that do not need to explicitly exist in a database [5]. However, traditional supervised approaches rely on vast historical datasets to make effective predictions in complex environments. Furthermore, these models are still susceptible to zero-day vulnerabilities, where the attack has no "learned" patterns found in the historical data.

Recent research has shown the success of RL in overcoming these deficiencies by allowing an agent to learn through direct environment interaction [6, 7]. This has proven beneficial in ACO; however, these agents start as "bare-bone" models that have to be trained from scratch. This is inefficient as the agent will initially be just as likely to choose an unfavorable action as a favorable one - it will inevitably make poor decisions before it can learn to select optimal ones. As a result, more training is needed before the agent can converge on an optimal policy.

Furthermore, the knowledge of these agents is only limited to the reward signals generated by the environment, making the development of transferable agents a challenge. There are two possible solutions to overcome this:

1. Create a perfect environment that represents all possible states and transitions that can be encountered in the cyber domain; or
2. Create an RL agent that can leverage an external source with "real-world" knowledge to assist in the decision-making process.

.

This research focuses on the latter of these two options. It should be noted that the first option is considerably challenging due to the complexities and rapidly evolving nature of cybersecurity.

The real-world knowledge is provided by an LLM for this study. An LLM's ability to recognize patterns in language is particularly helpful in cybersecurity, where samples are generally textual by nature (e.g., log files, event messages,

etc). Integrating an LLM into the pipeline enables the RL agent to leverage this knowledge from the beginning, optimizing its decision-making capability.

The motivation behind this thesis is to demonstrate the advantages of integrating an LLM, trained in the cybersecurity domain into the RL process for ACO. The RL agent can augment its decision-making capabilities by directly leveraging the LLM's knowledge of cybersecurity threat response [8, 9, 10] and pattern recognition [11]. By demonstrating the positive impact of LLM integration with the CybORG environment [2], this thesis highlights the potential real-world implications. CybORG is the cybersecurity environment used in this study, which is described in detail in Chapter 2.

1.2 Statement of Deficiency

Current RL algorithms employed in ACO require substantial training before converging onto an optimal policy. Initially, RL agents assign equal weights to all actions, meaning they are just as likely to select unfavorable actions as favorable ones during the early stages of training. In cybersecurity, the consequences of performing undesired actions can be severe, potentially resulting in the compromise of entire IT systems. This also leads to longer training periods, during which the agent must learn which actions to take based solely on the feedback signals from the environment.

Furthermore, the data used for responding to cybersecurity incidents is typically textual (e.g., logs, events, etc) and intended for human operators. RL agents require this information to be mapped into a numerical state space to extract meaningful patterns. However, this feature engineering process may miss critical information that is necessary for making informed decisions.

1.3 Aim

The aim of this thesis is to augment decision-making in ACO by integrating external knowledge in the form of an LLM into the RL pipeline, in order to reduce the training time required for an agent to converge to an optimal policy.

To evaluate the success of the thesis, an optimized RL agent is created to serve as a baseline. The performance of the baseline is then compared directly against the LLM-guided agents. Specifically, the success of this thesis is measured by:

- How quickly the LLM-guided agent converges onto an optimal policy compared to the baseline. This convergence is considered successful if it occurs more rapidly than the baseline in terms of timesteps. A

timestep is a single interaction in which the agent performs an action in the environment and receives both a reward and next state in response.
- How accurate the LLM-guided agent is within the RL environment. The evaluation of the agent's accuracy is considered successful if its policy yields equal or better performance than the baseline in terms of reward signals.
- The initial performance of the LLM-guided agent at the start of training in terms of the environment's reward signals. The initial performance criteria is considered successful if the LLM-guided agent outperforms the baseline early on, demonstrating the ability to learn without having to perform obviously unfavorable actions.

1.4 Research Activities

The following activities were performed for integrating an LLM into the RL pipeline for ACO. These are discussed at length in Chapter 4.

1. **Selecting an LLM**. Existing open-source LLMs pretrained in cybersecurity were evaluated using a dataset of manually created question and answer pairs. These pairs were contextually relevant to CybORG, containing states that closely resemble those returned directly from the environment. The validation of responses was initially facilitated through BERTScore [12]; however, manual evaluation across the responses was ultimately used to determine the best LLM.
2. **Environment Modification**. The CybORG environment was modified to better reflect realistic cybersecurity conditions and support RL training. These modifications include:
   - Modifying the action-space of the RL agent; and
   - Adjusting the reward signals returned by CybORG; and
   - Changing how the environment's raw output is transformed into engineered features.
3. **Baseline Agent Development**. This activity was done in parallel with environment modifications due to their interdependencies. An agent was required to validate the environment modifications, and the environment modifications were necessary to enable more realistic training for the agent. Both value-based and policy-based RL agents were evaluated during this phase.
4. **Developing a Teacher-Guided Algorithm**. Multiple algorithms (and combinations thereof) for integrating a teacher into the RL pipeline were developed and assessed to determine which is the most effective for CybORG. These were based on existing work [13, 14, 15, 16, 17, 18, 19].

5. **LLM Integration into the RL Pipeline**. This involved integrating the best-performing LLM with the best-performing teacher-guided technique.
6. **Evaluation of Results**. The LLM-integrated RL pipeline was evaluated against the baseline agent. The primary metrics used in this evaluation were the reward signals outputted by the environment and the convergence speed measured in timesteps.

1.5 Organization

The remainder of this thesis is organized as follows. The **Background** chapter provides all relevant information on LLMs, RL and ACO. The **Related Work** chapter presents previous research in this field. The **Methodology** chapter elaborates on the research activities. The **Evaluation** chapter describes how well techniques performed, and the rationale behind their implementation. The **Conclusion** summarizes the study's contributions, discusses its limitations and presents ideas for future work.

2 Background

2.1 Large Language Models

Fundamentally, a Large Language Model is a Deep Neural Network (DNN) designed to recognize patterns in language [11]. The vast number of parameters and the unique transformer architecture [20] enable LLMs to identify complex patterns in language, and generate contextually relevant responses for a wide range of Natural Language Processing (NLP) tasks.

2.1.1 Self-Attention

A key component of the transformer architecture is the self-attention mechanism [20], which computes a vector signifying how each token in the input sequence relates to others. This vectorized representation allows the model to process input sequences in parallel, eliminating the need to parse tokens sequentially, as required in earlier architectures such as the Recurrent Neural Network (RNN) [21].

Figure 2.1 illustrates how self-attention is applied to the phrase "AI is cool". As shown, the output (A_x) is computed using the current token (X_i) and all its predecessors ($X_i, X_{i-1}...X_0$). Fundamentally, this is a weighted sum [22]:

$$\mathbf{a}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j$$

where α_{ij} is the weight that token j has on the attention score for token i.

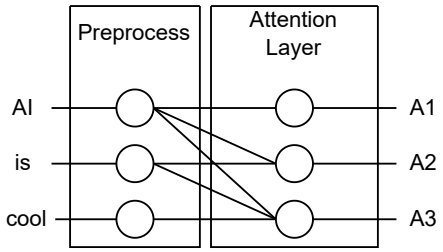


Figure 2.1: Overview of the self-attention mechanism [20]. The original tokens are shown on the left with their associated attention scores shown on the right. Specifically, this represents masked self-attention, where each token's score is calculated using only itself and preceding tokens.

In practice, the transformer model uses three weight matrices to project each token, effectively assigning three distinct "roles" [22]:

- **Query**. A matrix representing the input as the current element being compared to the preceding ones. It is calculated using $q_i = x_i W^Q$ where W^Q is the query weight vector.
- **Key**. A matrix representing the input as a preceding input being compared to the current element. It is calculated using $k_i = x_i W^k$ where W^k is the key weight vector.
- **Value**. A matrix representing the input as the value of a preceding element that gets weighted to calculate the output of the current element. It is calculated using $v_i = x_i W^v$ where W^v is the value weight vector.

An attention score is then calculated by taking the dot product of each query and key value. This score is scaled down by the root of the dimensionality of the key vector [22] to discourage sporadic updates:

$$score(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$$

A softmax activation function is then applied to convert these scores into the weights that determine how much token i attends to token j:

$$\alpha_{ij} = softmax(score(x_i, x_j))$$

Finally, a weighted sum is applied to compute the self-attention output for token i, using the value vector:

$$a_i = \sum \alpha_{ij} v_j$$

Having a vectorized representation of how each element in an input sequence relates to each other allows the model to perform computations in parallel - it does not need to wait for the previous token, as is the case for RNNs. This greatly improves training efficiency and increases the model's ability to capture contextual information across entire input sequences. However, while more efficient, this approach increases memory and CPU requirements due to its parallelized nature.

2.1.2 LLM Architecture

The preceding subsection provided an overview of the self-attention mechanism; however, it did not explain how this fits into the full transformer architecture. The original transformer architecture consists of the following components [20]:

1. **Tokenization**. Before a sequence is fed into the model, it is tokenized - split into smaller units that can be easily mapped into a numerical feature space. These units are typically the words from the prompt, but can be as small as individual characters.
2. **Embedding Layer**. This layer converts the tokens into continuous, numerical representations, enabling the neural networks to recognize and process patterns across tokens.
3. **Positional Encoding**. This adds additional information to the embeddings that represents in which order each token appears. This is critical in language where the order of words can completely alter semantic meaning. Positional encoding enables the transformer to capture these sequential dependencies, even though processing occurs in parallel.
4. **Encoding**. The encoder creates a set of hidden representations to capture contextual relationships among the input tokens. It consists of two primary layers: a self-attention layer and a feed-forward layer. Unlike the masked self-attention shown in Figure 2.1, the encoder uses bidirectional self-attention, where the tokens' attention scores are calculated using both preceding and future tokens. This enables the model to form a complete representation of the input. The feed-forward layer enables the model to form complex, non-linear dependencies between features, essential for extracting patterns from the complex semantics associated with language. The output of the encoder is a vector for each token that reflects its relationship to all other tokens in the input sequence.
5. **Decoding**. The decoder takes the encoded output and uses it to predict the next token, comparing this prediction to the actual next token to

compute the loss. For example, if the input sequence is "AI is cool", the decoder will use the hidden state for "AI" to predict the next token ("is"). The actual prediction is a vector of logits whose size equals the number of possible tokens. Each of these logits is passed through a softmax activation function to create a probability distribution across the tokens (stochastic output). To prevent the decoder from accessing future tokens, masked self-attention is applied. This is implemented by setting all elements greater than i to $-\infty$, causing them to be effectively 0 after passing through the softmax activation function, ensuring they do not influence the output [20].

Figure 2.2 depicts the above process for which a transformer predicts the next token from the input: "The coolest field is".

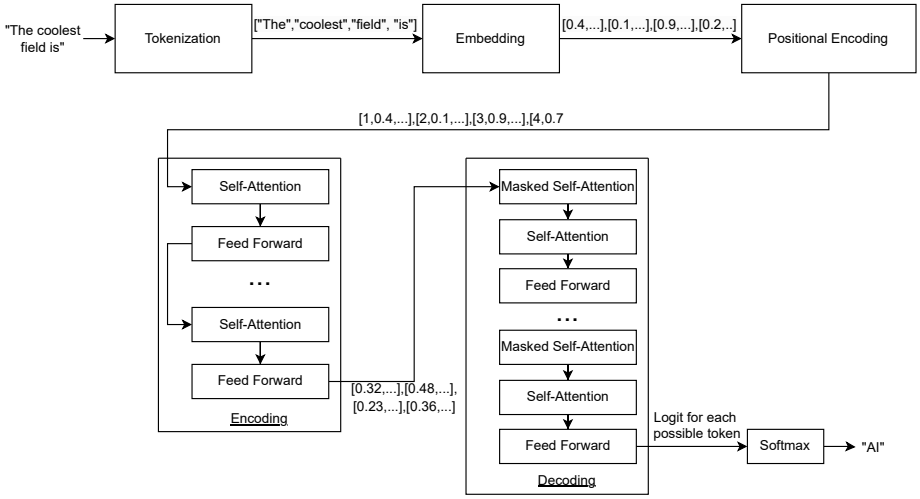


Figure 2.2: Illustration of the components in the transformer architecture and the order in which they are used to generate a prediction. This figure is kept at a higher level than the original transformer architecture [20] to emphasize the core functionality and components.

It should be noted that the decoder and encoder consist of multiple layers. This allows the model to consider different "meanings" of the input [20]. For example, one layer might be focused on the grammar of the input sequence whereas another might be focused on the intent.

The process described above outlines the original transformer architecture, which is used for encoder-decoder type models. The T5 introduced by Raffel et al. [23] is among the most popular family of LLMs that use this architecture.

There are two other primary types of LLMs with minor deviations from the original transformer architecture:

1. **The decoder-only model** [24]. This type of model does not have the encoder block shown in Figure 2.2. It is designed for auto-regressive tasks such as text generation, where the decoder predicts the next token based on previously generated tokens and the input sequence. Masked self-attention is essential to ensure that future tokens are not used for its prediction. The Generative-Pretrained Transformer (GPT) is the most popular family of LLMs that use this model [25].
2. **The encoder-only model** [24]. This type of model does not have the decoder block shown in Figure 2.2. As a result, there is no masked self-attention, permitting the model to use both past and present tokens in the input sequence for making a prediction. This bidirectional representation of the input makes it ideal for classification tasks such as sentiment analysis. These models are not suitable for text generation tasks because they lack the auto-regressive mechanism of a decoder and are instead designed to *encode* the input. The Bidirectional Encoder Representations from Transformers (BERT) is the most popular family of LLMs that use this model [26].

It should be noted that the majority of recent achievements in LLMs [27, 28] are not due to modifications in the transformer architecture, but to an increase in the amount of data and parameters used to train these models.

2.1.3 Pretraining

The training of an LLM generally occurs in two phases: the pretraining phase and the fine-tuning phase.

The pretraining phase requires the most time and resources. This involves collecting vast amounts of data (up to petabytes) and feeding it to the model. The model then adjusts its parameters to predict tokens based on the patterns identified in the data it has been exposed to. The goal of this phase is to have a model that can recognize patterns in language and use this contextual representation to predict the next token (or masked token) in a sequence. This learning is considered "self-supervised", a subcategory of unsupervised learning where the model generates its own labels from the input data. For decoder-only models, the LLM masks the last token in the input sequence and uses all preceding ones to make a prediction. It then compares this prediction with the real last token to calculate the loss, which is used to adjust the model's parameters to improve performance (typically done in batches). An identical

concept is applied to encoder-only models; however, a random token is masked instead of the last one, and it uses all tokens in the input sequence to make its prediction. The pretraining phase can take weeks and cost millions of dollars for larger LLMs [29].

2.1.4 Fine-Tuning

Once the pretraining phase is complete, and the LLM can predict subsequent tokens, fine-tuning can begin. Unlike the self-supervised pretraining phase, fine-tuning involves conducting targeted training focused on specific tasks rather than just predicting the next token based on contextual patterns in language. Examples of tasks that an LLM can be aligned to include: answering questions, code generation and sentiment analysis. The following subsections discuss the various ways alignment can be achieved after pretraining.

Supervised Fine-Tuning

Supervised Fine-Tuning (SFT) is standard supervised training that involves creating a dataset of samples and their corresponding labels [30]. As in traditional supervised learning, the LLM adjusts its parameters to minimize the loss between the true labels and its predictions. While the dataset constructed for this has to be fairly substantial to get the LLM to adjust its output, it is still significantly smaller than the amount of data used for the pretraining phase. SFT is generally combined with other fine-tuning methods to align LLMs to a specific task.

Reinforcement Learning with Human Feedback

Reinforcement Learning with Human Feedback (RLHF) is a fine-tuning technique typically conducted after SFT [31]. The goal is to incorporate human feedback into the model's learning process to generate more relevant outputs. This process involves presenting multiple (generally two) outputs from the same prompt and having a human select the preferred one. These outputs are then fed into a reward model that calculates a corresponding score for each - ensuring that the preferred outputs receive higher scores than the unpreferred ones.

Once the reward model has been trained, it is used to provide numerical scores for new outputs generated by the LLM. RL is then used to adjust the LLM's parameters to maximize the score given by the reward model.

Direct Preference Optimization

One of the disadvantages of RLHF is that it requires training a new neural network to generate the reward signals used to fine-tune the LLM. Direct Preference Optimization (DPO) addresses this by *directly* aligning the LLM based on human feedback [32]. Similar to RLHF, multiple responses are generated from the same input, and the preferred one is selected. The loss is then computed by directly comparing the raw probably distributions (logits) of the preferred and non-preferred responses, effectively bypassing the RL process. The loss is calculated using Binary Cross-Entropy (BCE) - the same loss function used in binary classification. The advantages of this approach include: simpler implementation, increased computational efficiency and more stable updates [32]. Figure 2.3 shows the differences between using RLHF and DPO for fine-tuning an LLM.

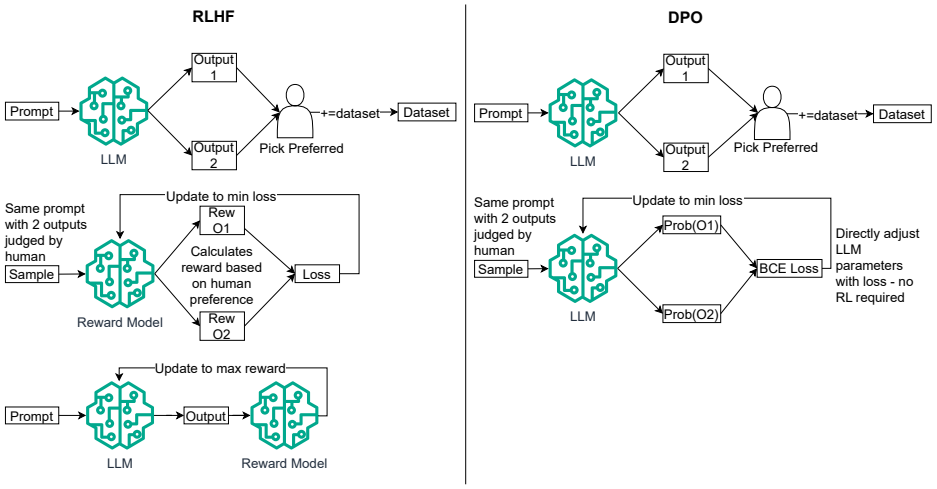


Figure 2.3: Difference between Reinforcement Learning with Human Feedback (RLHF - left) and Direct Preference Optimization (DPO - right) [31, 32]. DPO has no separate reward model and generates its loss directly from the preferred and unpreferred outputs.

Full Fine-Tuning vs Partial Fine-Tuning

During fine-tuning using one of the above techniques, the LLM's parameters are adjusted to optimize its output for a specific task. There are two methods of adjusting these parameters:

1. **Full fine-tuning**. In full fine-tuning, **all** of the LLM's parameters are

updated. The advantage of this approach is that the original LLM can be significantly modified; however, this comes at the cost of increased training time. Another disadvantage is that the LLM could lose some of the knowledge it gained from the pretraining phase.

2. **Partial fine-tuning**. In partial fine-tuning, only a subset of the LLM's parameters is adjusted. This allows the LLM to retain more of its baseline knowledge and improves training efficiency. The most common partial-fine tuning technique is Parameter-Efficient Fine-Tuning (PEFT) [33], where parameters are kept frozen during training. Typically, the focus is on adjusting parameters in the later layers to have the greatest effect on the output; however, any parameter can be adjusted.

2.1.5 Prompt Engineering

The fine-tuning techniques discussed above optimize an LLM by physically changing the model through adjusting its parameters. In contrast, with prompt engineering, the model remains static and the input (the prompt) is modified to create an output that better aligns with the user's intent [34]. Providing clear instructions, specifying constraints and including examples are some ways users can improve their prompts to achieve more effective and accurate outputs from an LLM.

Since prompt engineering does not modify the underlying LLM, it is most effective for models that already have some knowledge on the task's domain. The following subsections discuss various methods for optimizing prompts.

X-Shot Learning

X-shot learning modifies the prompt to include examples that illustrate the user's intent for completing a task [35]. The X signifies the number of examples included with the prompt. A simple example of one-shot learning for cybersecurity is:

> **Select the action to defend my system given the following example:**
> **State:** Outbound traffic from Host A to known malicious IP
> **Action:** Isolate Host A
> **State:** `Stuxnet.exe` file present on Host B
> **Action:**

By providing explicit examples, this method guides the LLM's output to better-align with the user's intent.

Chain-of-Thought Prompting

This technique involves asking the LLM to provide steps explaining how it reached its answer [34]. One considerable benefit of this approach is the increased transparency between the user and the LLM's decision-making process. This has also shown general improvement in the quality of the generated outputs [36, 37]. An example of chain-of-thought prompting for cybersecurity is shown in Figure 2.4. The ChatGPT-4o LLM was used for this example [38].

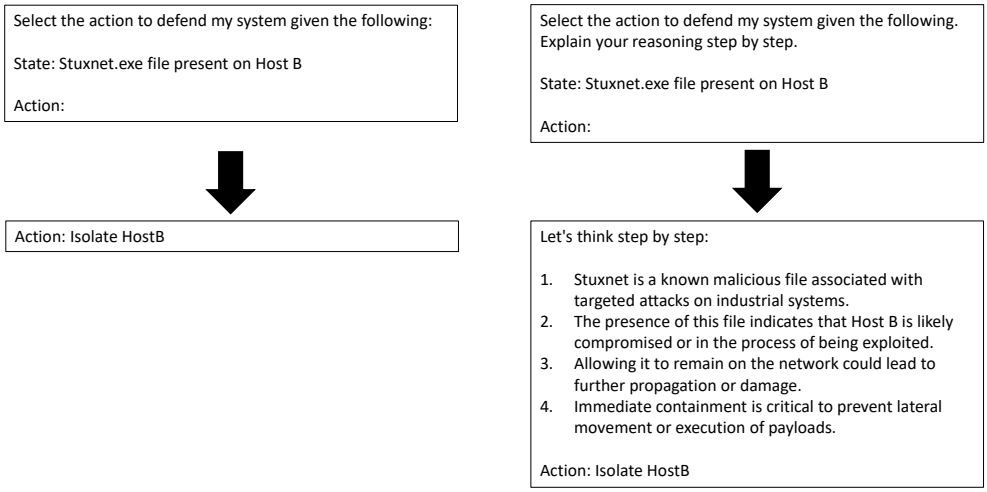


Figure 2.4: Comparing the results of a prompt that uses chain-of-thought reasoning (right) against a prompt that does not (left).

This step-by-step reasoning can also be achieved by dividing the task into multiple sub-prompts, enabling the model to gather additional context from previous prompts to enhance the quality of future ones.

Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) involves leveraging existing knowledge in conjunction with the model's learned parameters to complete a task [39]. This requires access to an external data source with domain-specific knowledge related to the task. This external knowledge is incorporated into the prompt to guide the model to generate a response that better aligns with the user's intent. By accessing the external knowledge base, the model can significantly reduce hallucinations and outdated responses [39].

Conceptually, RAG is few-shot learning, but with the examples being added automatically, transparent to the end-user. Figure 2.5 shows the process of RAG: **R**etrieving the information from the external database, **A**ugmenting the current prompt and **G**enerating the optimized response.

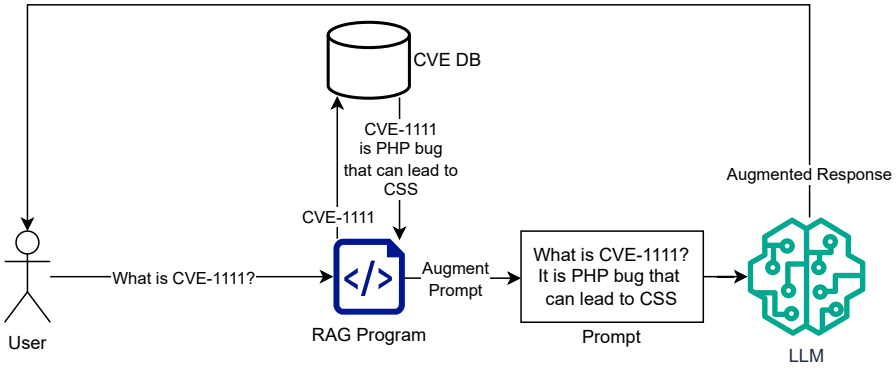


Figure 2.5: Overview of the Retrieval-Augmented Generation (RAG) process [39]. An external database is queried to augment the prompt and provide the Large Language Model (LLM) additional context to generate a response.

2.2 Reinforcement Learning

Reinforcement learning is its own machine learning paradigm, alongside supervised and unsupervised learning [40]. In supervised learning, each sample in the dataset has a corresponding label, allowing for direct calculation of loss to adjust a model's parameters. In unsupervised learning, there are no labels, and the model groups samples based on identified patterns.

However, RL is fundamentally different as it does not rely on a predefined dataset [41]. Instead, the agent directly interacts with the environment by taking actions. The environment then outputs a state and corresponding reward that the agent can use in a trial-and-error like fashion until it can learn to select the actions that maximize the cumulative reward. The following terminology is used throughout RL:

- An **agent** is the entity that actually interacts with the environment by selecting actions. This generates a new state and corresponding reward that it uses to optimize its actions.
- The **environment** is the "world" that the agent interacts with. It generates new states and rewards based on the actions that the agent

takes.

- **States** are current scenarios or situations of the environment. An example of a state in the cybersecurity context is a network that has just been exposed to malware. The agent uses the state to ultimately decide what action to take.
- **Rewards** are scalar values returned by the environment to determine how favorable an action was.
- The **policy** is the strategy that the agent uses to decide what action to select.

A high-level overview of the RL process is shown in Figure 2.6 [41].

Formally, an environment that outputs the next state and reward based on a given action is known as a Markov Decision Process (MDP).

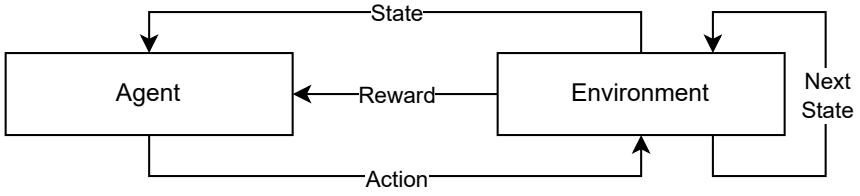


Figure 2.6: High-level overview of the Reinforcement Learning (RL) process [41]. An agent interacts with the environment by selecting an action, which generates a reward signal and next state.

2.2.1 Model-Based vs Model-Free RL

There are two primary types of RL: model-based and model-free RL [42]. In model-based RL, the agent uses the environment's output signals to construct its own representation of it. It can then use this representation to predict future states and rewards without direct environment interaction. In model-free RL, the agent learns directly from the the environment's signals without constructing its own internal model of it. For this study, all RL will be model-free. This choice is due to the complexities and dynamic nature of the cybersecurity domain, where the environment cannot be realistically represented by an agent's internal model. For the remainder of this thesis, model-free RL will be referred to simply as RL.

2.2.2 Policy Convergence

In RL, there are two main functions an agent can use to converge onto an optimal policy (i.e., its strategy for selecting actions): the value function and

the policy function [43]. The value function estimates the expected return from a state when following a given policy. There are two types of value functions:

- **State-value function**. This estimates the expected reward starting from a specific state and following the same policy until the end of the episode.
- **Action-value function**. This estimates the expected reward for taking an action in a given state and then following a policy until the end of the episode.

The policy function, in contrast, is fundamentally different. Rather than predicting the expected reward of an action in a given state, it directly returns the action an agent should take. There are two primary ways to achieve this:

- **Deterministically** where the same state will always map to the same action. This is conceptually similar to nested if statements in programming.
- **Stochastically** where there is an element of randomness introduced into the selection of an action. Typically, the output of a stochastic policy function is a probability distribution across all possible actions. While the action with the highest probability is the most likely to be selected, it will not always be the one chosen.

2.2.3 Monte Carlo and Temporal Difference

The reward signal received from the environment is the main indicator that an RL agent uses to determine how favorable an action is. However, the immediate reward of a step cannot be used as the **only** indicator for how favorable an action is for a given state, as this approach completely ignores potential future rewards. For example, if winning a chess game requires sacrificing the queen, the immediate reward will be negative; however, the future reward will be substantial (getting checkmate). Future rewards must be accounted for when selecting an action in a given state. There are two main methods in RL to account for future rewards: Monte Carlo (MC) and Temporal Difference (TD) [41].

In MC, the agent uses the total reward collected over an entire episode to estimate the current value of a state. Future rewards are given less weight by applying a discount factor that decays exponentially. This is primarily because immediate rewards are more certain than future rewards; however, there are also directly applicable scenarios for this - such as in intrusion detection, where an immediate response yields more reward than a delayed one, since early intervention can prevent further propagation or escalation.

The main advantage of MC is that the value function is updated based on the actual returns from following a given policy throughout an episode. However, this comes at the cost of requiring the agent to wait for an entire episode before updating its parameters. Furthermore, the unbiased estimate results in higher variance, as the agent always uses the full return from an entire episode for training.

In TD, the value of a state is updated at each timestep by taking an estimate of the future rewards using the current value function. This bootstrapping technique allows the agent to learn without waiting for an episode to complete, resulting in quicker training and lower variance, as updates are made more frequently. However, this also introduces bias, as the values are estimates and will never be as accurate as methods that use complete trajectories, such as MC.

2.2.4 On-Policy vs Off-Policy RL

As described above, an RL agent learns by directly interacting with an environment to learn the best actions to take (i.e., an optimal policy). There are two primary ways the agent can converge to an optimal policy: on-policy and off-policy learning [43].

Off-Policy Learning

In off-policy learning, the agent can learn from data that was generated using *different* policies. This is generally more sample-efficient, as the agent can incorporate past data into its learning process. This past data does not have to be generated by the agent itself, but can be from any source, such as other agents. This is especially important in environments where direct interactions are costly or reward signals are sparse. This method is sample-efficient because the agent does not require new data for each update. The primary disadvantage of off-policy learning is instability, as the agent's parameters are adjusted using data generated from a different policy than the one used to interact with the environment.

Furthermore, the output of an off-policy approach is generally an action that yields the highest Q-value, rather than a distribution across multiple actions. This requires manual strategies to balance exploration versus exploitation, ensuring the agent does not get stuck in a local minimum.

One of the most notable off-policy methods for RL is Q-learning [44]. In Q-learning, a table is used to record values for every action in every state (Q-values). The Q-values are initialized to 0 and updated as the agent interacts

with the environment. TD is typically used, where the Q-value is updated based on the immediate reward and the Q-value of the next state-action pair (multiplied by a discount factor). Multiple iterations are run to ensure that the Q-values incorporate all potential future rewards. For example, if action A in state A yields an immediate substantial reward, but results in losing a game, the Q-value will initially be large for that given state. As updates are performed, the agent will eventually encounter the action that leads to losing the game, which will be backpropagated, greatly reducing the once favorable Q-value for performing action A in state A.

To decide what action to take, the agent can always select the highest Q-value for its current state; however, this eliminates exploration of the environment and the agent will continue to exploit what it perceives to be the most favorable action. The most common technique to prevent the agent from continuously exploiting a small subset of actions is the epsilon-greedy approach. This means that for each step in the environment, the agent has an *epsilon* probability of choosing a random action instead of acting greedily by selecting the action with the highest Q-value. Generally, this epsilon value is decayed over time to ensure that the probability of the agent acting randomly is inversely proportional to its ability to select favorable actions.

In most environments, having a table that contains a mapping between every possible state and every possible action is not feasible. Instead, the Q-table can be replaced by a DNN. This is referred to as a Deep Q-Network (DQN) [44]. Instead of looking up the value directly in the table, the agent passes the state to the DNN, which returns the Q-values for all the actions. The same epsilon-greedy approach can be applied to ensure the agent explores the environment while still selecting the best possible action.

On-Policy Learning

In on-policy learning, the agent learns from data that was generated by the same policy. In other words, the policy the agent uses to decide what action to take is the same policy used to adjust its parameters to obtain the maximum reward. This typically results in more stable training as the agent's parameters are updated directly based on data collected by the current policy. Typically, the output of an on-policy agent is stochastic, which naturally balances exploration and exploitation during the learning process.

The disadvantage of on-policy learning is that it is less sample-efficient, as samples generated from past policies are discarded. Additionally, because the agent learns only from data collected by its current policy, it is difficult to incorporate data generated from other sources into the learning process.

This poses challenges for the emerging multi-agent methods [6] used to solve complex problems.

One of the most effective on-policy methods for RL is Proximal Policy Optimization (PPO) [45]. PPO directly updates the policy function using stochastic gradient ascent, meaning that there is inherent randomness in the policy's output, and the parameters are adjusted to maximize the reward (gradient ascent). To ensure stable training, PPO applies a clipping mechanism to prevent policy updates from being too drastic. PPO consists of two neural networks:

- An **actor** that outputs a probability distribution across all possible actions.
- A **critic** that outputs a scalar estimate of a state's return. PPO does not use the output of the value function directly, but leverages a Generalized Advantage Estimate (GAE), which balances TD and MC for accounting for future rewards using a smoothing parameter.

Both PPO and DQN are explored in this study for selecting a baseline RL agent. Details of their implementation are discussed further in Chapter 4.

2.3 Autonomous Cyber Operations

Autonomous Cyber Operations (ACO) refers to conducting cybersecurity tasks with minimal or no human intervention. This approach has significant benefits due to the increasing scale and complexity of cyber threats, where manually defending systems is no longer feasible. The goal of ACO is not to replace cyber operators, but to work alongside them to enhance the effectiveness of cybersecurity.

Previous work [7, 36, 6, 46] has shown that RL is an effective way to achieve ACO. This requires an environment that is interactive and provides meaningful signals. Such an environment must reflect the complexities of real-world cybersecurity, to ensure the agent is training on relevant data. Typically, this environment is simulated or emulated.

- In an **emulated environment**, a physical system is created to mirror the real world. A virtual network with actual virtual machines would be an example of an emulated environment for cybersecurity. The main drawback is the high resource usage and the difficulty of replicating complex environments.
- In a **simulated environment**, the real world is not physically replicated; instead, a model of it is created. This approach is less resource intensive,

more scalable, and easier to implement than an emulated environment; however, it may not be as realistic.

The following subsection discusses CybORG, one of the most popular environments used in previous work for ACO. The decision for selecting CybORG for this study is discussed further in Chapter 3.

2.3.1 CybORG Overview

CybORG [47] is a platform created by The Technical Cooperation Program (TTCP) for developing RL tactics in a cybersecurity setting. TTCP is a collaborative forum between UK, USA, Canada, Australia and New Zealand, comprising of roughly 1,000 defense scientists [48].

CybORG releases various open-source challenges (Cage Challenges) that contain scenarios representing a blue team's system. The purpose of these challenges is to provide a realistic environment for implementing RL techniques for cybersecurity. The Cage Challenges typically support both simulated and emulated modes; however, due to the high resource costs and lack of scalability associated with emulation, simulation was used for this study. Since these environments are simulated, every component is represented by a Python object. For example, instead of having a VM that contains files, connections and processes, an object with instance variables is used to represent the state of the host. As the scenario progresses, these variables are modified to simulate the actual changes that would occur in real life.

At the time of writing, there are four Cage Challenges:

1. **Cage Challenge 1** [49] is a scenario designed to defend a simple network from an attacker.
2. **Cage Challenge 2** [50] is the same as Cage Challenge 1, but implements more frequent reward signals and expands the action space.
3. **Cage Challenge 3** [51] introduces a new scenario using autonomous drones instead of standard workstations.
4. **Cage Challenge 4** [52] is the same as Cage Challenge 1 and 2, but introduces additional complexities. For example, the number of hosts in the network is randomized with each run. This challenge is designed for multi-agent scenarios.

The multi-agent support that Cage Challenge 4 offers is not applicable to the objectives of this thesis, and could introduce complexities in the debugging and development process. Moreover, although the autonomous drone-based network in Cage Challenge 3 is interesting, it does not represent the typical cybersecurity scenario. Given these factors, Cage Challenge 1 and Cage

Challenge 2 better align with the goals of this thesis. Since Cage Challenge 2 is an improvement on Cage Challenge 1, it is the challenge used in this thesis.

2.3.2 Cage Challenge 2

Cage Challenge 2 [50] is designed to support blue ACO by providing a network that a red agent is attacking. The goal is to create a blue agent capable of successfully defending the network against the attacker. The challenge also includes green agents, which represent standard network users. Cage Challenge 2's original simulated network contains three subnets:

1. A **user subnet** that the attacker always has an initial foothold in; and
2. An **operational subnet** containing the operational server the attacker ultimately wants to compromise; and
3. An **enterprise subnet** that connects the user and operational subnets.

Figure 2.7 shows the original network used for Cage Challenge 2. The attacker's goal is to compromise the operational server by moving laterally from the user subnet [50].

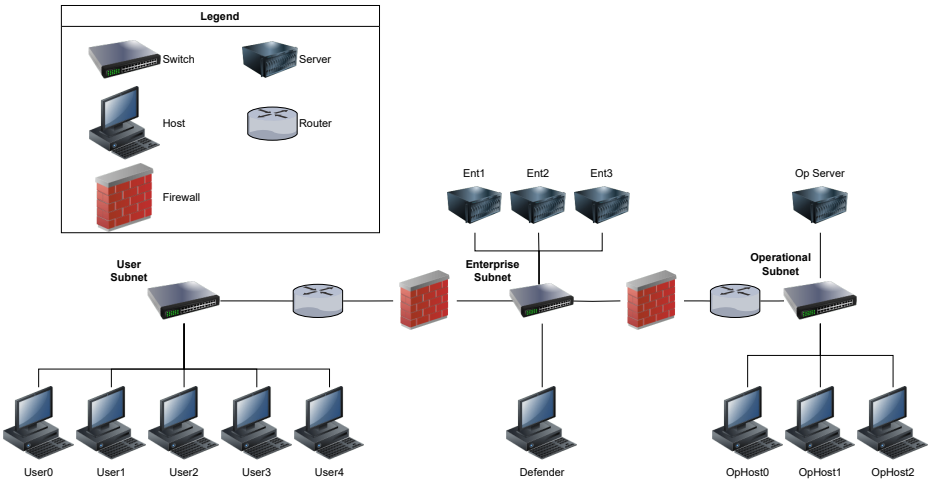


Figure 2.7: Overview of the simulated network used for the Cage Challenge 2 scenario [50] Ent and Op are short for Enterprise and Operational, respectively.

Environment Interaction

Agents interact with the CybORG environment by selecting actions. These actions are Python objects whose impact is simulated by modifying attribute

variables associated with the overall scenario (also represented by Python objects). At every timestep, each agent performs an action on the environment, starting with the blue agent, followed by the red agent. The blue agent's action space is designed to defend the network, whereas the red agent's action space is designed to compromise it. The blue agent's action space can be categorized into three types [47]:

- **Reconnaissance actions**. These include the monitor action to gather information about the overall environment and the analyse action to obtain additional information about a specific host.
- **Restorative actions**. These include the remove action to eliminate an attacker's user-level presence and the restore action to return a host to its original state (at a penalty).
- **Deception actions**. These include actions to create fake processes to slow down the attacker.

Similarly, the red agent's action space can be described as:

- **Reconnaissance actions**. These include actions to find hosts on a network and discover vulnerable services running on a specific host.
- **Exploitation actions**. These include actions to exploit a vulnerable service to gain a foothold and a privilege escalation action to escalate to root access on a host.
- **Effects**. This includes an impact action that simulates the attacker executing their objective, such as data exfiltration or disruptions to network functionality. The goal of the attacker is to execute this action on the operational server.

The blue agent's action space is discrete, meaning the agent can always perform the same action on every target. In contrast, the red agent's action space is dynamic, because many actions depend on the knowledge gained from previous ones (e.g., the attacker must first discover a service before exploiting it). Figure 2.8 is taken directly from Cage Challenge 2's GitHub repository and shows the effect of blue and red actions on the environment's state [50].

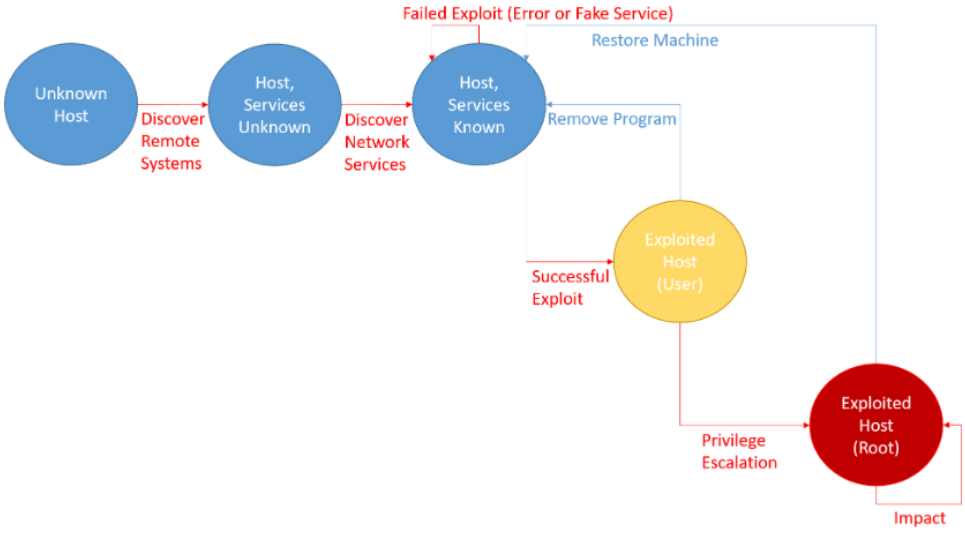


Figure 2.8: Diagram depicting the effect that actions available to the red and blue agents have on the environment [50].

Similar to all RL environments, executed actions in CybORG yield a reward signal. This reward is a non-positive (i.e., zero or negative) float that is calculated from the current state of the environment. For example, if a host is compromised, the blue agent will receive a lower reward each timestep compared to when the host is not compromised. The reward signal also depends on the last actions performed by both the red and blue agents. The blue agent is penalized for selecting the restore action, while the red agent is rewarded if it successfully executes the impact action on the operational server.

The environment includes two red agents whose goals are to exploit the network:

1. A **meander agent** that randomly selects available actions; and
2. A **B-line agent** that consistently follows the fastest path to reach the operational server. It does this by following a trajectory of predefined actions, simulating an attacker with inside knowledge of the network. This is the agent used predominantly throughout this study.

Wrappers

In Cage Challenge 2, the observations returned are Python dictionaries containing key-value pairs with information related to each host (e.g., its processes

and files). Cage Challenge 2 includes a wrapper class (BlueTableWrapper) that converts these to discrete states. It does this by saving the default files and processes each host has at the start of an episode, and comparing them against those in the agent's observation space. If the observation contains processes or files that are not part of the baseline image, they are flagged as potentially malicious and stored in a list. These lists are then parsed to extract a compromised and activity state for each host, where the activity state signifies the red agent's current activities on the host and the compromised state reflects the level of access the red agent has achieved. The activity state can be:

- **Exploit** if the number of connections is greater than 2 for a single port.
- **Scan** if there are more than 0 connections across 1 or more ports.
- **None** if there are no new connections, or if the connection includes a recognized service name (CybORG simulates recognized services by including *Service Name* in the connection string).

The compromised state can be:

- **Privileged** if a file with a density score (score signifying how likely a file is to be malicious) greater than 0.9 exists that is not part of the baseline image.
- **User** if the corresponding availability state (from above) is set to *exploit*.
- **Unknown** if the remove action was performed on the host that was previously identified as compromised. This is because the remove action only works on user-level processes (i.e., simulating uncertainty about whether the adversary has a system-level presence).
- **No** if there are no new files contained in the observation space.

These states are then converted to numerical features that can be processed by an RL agent. The state values are mapped to binary encodings of 1s and 0s for each host, in which the activity and compromised state are each represented by 2 bits. These are aggregated into an array, where the index of the encoding corresponds to that particular host. This is the feature space that is fed into the RL agent. Figure 2.9 shows the default mapping between the hosts' states and their numerical representation.

Activity Mapping	
None	[0,0]
Scan	[1,0]
Exploit	[1,1]

Compromised Mapping	
No	[0,0]
Unknown	[1,0]
User	[0,1]
Privileged	[1,1]

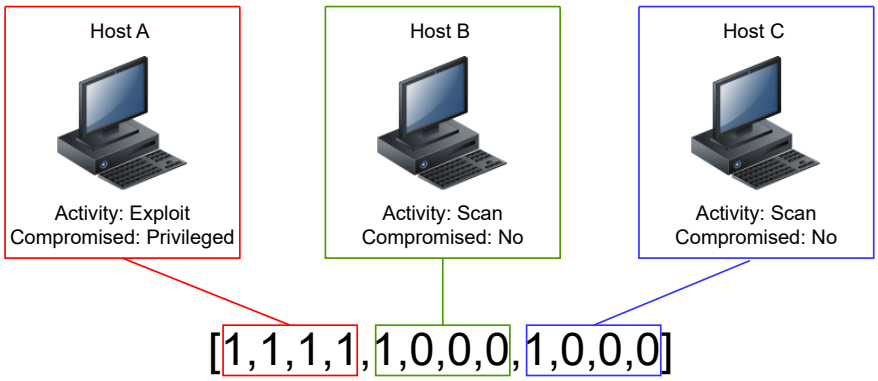


Figure 2.9: Default feature mapping for Cage Challenge 2. Each host is assigned an activity and compromised state, each represented by a 2-bit value.

This section covered the high-level functionality of CybORG's Cage Challenge 2, but did not include the underlying complexities. Appendix A includes sequence diagrams illustrating all the objects and methods involved for standard functionality, such as performing a single step.

3 Related Work

The purpose of this chapter is to critique studies in the fields described in the background and synthesize insights into a new approach for enhancing current applications in Autonomous Cyber Operations (ACO). A comprehensive understanding of the relevant fields is essential for effectively identifying the limitations and potential improvements of existing work.

3.1 Teacher-Guided RL

One consideration for using traditional Reinforcement Learning (RL) is that agents start as "bare-bone" models that must be trained from scratch. This is inefficient because initially, the agent is just as likely to choose an obviously unfavorable action as a favorable one. As a result, it will inevitably make poor decisions before it can learn to select optimal ones, increasing the training time required to converge on an optimal policy.

M. Pfeiffer et al. [13] proposed a method to decrease training time and improve sample efficiency for mapless navigation. Their approach leverages a pretrained agent (i.e., the teacher) to create a dataset of state-action pairs used to initialize a student agent via standard supervised learning. The student then undergoes the RL process, ultimately surpassing the teacher's baseline performance. The main RL algorithm employed is Constrained Policy Optimization (CPO) [13], an actor-critic method where the actor outputs a probability distribution over possible actions and the critic returns a scalar indicating how favorable the chosen action is.

The methodology is as follows [13]: the pretrained agent generates a dataset of state-action pairs:

$$D = \{(s_i, a_i)_{i=1}^{n=1}\} \text{where } a_i \sim \pi^T(a \mid s)$$

where π^T is the teacher's policy used to generate actions a_i from the corre-

sponding states s_i. The new agent is then trained on this dataset:

$$\theta^* = \min_{\theta} \mathbb{E}_{(s,a)\sim D}\big[\mathcal{L}(\pi_\theta(a \mid s), \pi^*(a \mid s))\big]$$

where $\mathcal{L}$ represents the loss function comparing the agent's predictions to the teacher's labels. After initialization, traditional RL is used to further optimize the student:

$$\theta_{\text{RL}} = max_\theta \mathbb{E}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \gamma^t r_t]$$

where γ^t is the discount factor that controls how much future rewards contribute. The agent's goal is to maximize its cumulative reward across T timesteps.

This method successfully reduced training time for simulated navigation tasks of varying difficulty compared to a baseline CPO implementation. As expected, there was no noticeable improvement in the final policy - rather, the benefit was in faster convergence.

One limitation of this approach is that the teacher's feedback is generated in isolation from the environment. In dynamic and stochastic environments, such as those modeling cybersecurity, there is a risk that the environment will generate signals that are misaligned with the teacher's policy. The immediate transition from learning on a dataset generated in isolation by the teacher, to learning directly from the environment's signals could present challenges. The agent may become overly reliant on the teacher, potentially increasing training time rather than reducing it. Additionally, creating a comprehensive dataset that captures all possible edge cases for complex environments is challenging and requires significant effort.

A. Beikmohammadi and S. Magnusson [14] proposed incorporating the teacher's feedback within the RL environment rather than using it in a separate pretraining phase. This is achieved through reward shaping [14], where the teacher's recommendation is integrated into the environment's reward signal. Proximal Policy Optimization (PPO) [14] (the actor-critic method described in Chapter 2) is the RL algorithm used in this study. The influence of the teacher's feedback on the reward signal starts high, and gradually decreases as training progresses. Specifically, they proposed the following [14]:

$$\begin{aligned} R^e(s_t, a_t, s_{t+1}) = \beta(e) R^A(s_t, a_t, s_{t+1}) + \\ (1 - \beta(e)) R^T(s_t, a_t, s_{t+1}) \end{aligned}$$

where R^e is the reward signal the agent receives, computed as a weighted sum of the environment's original reward (R^A) and the teacher's reward (R^T).

The weight $\beta(e)$ decays over time, gradually reducing the teacher's influence as training progresses.

This study demonstrated increased training efficiency, showing faster convergence in the Random Walk, Optimal Temperature Control, and Coupled Four-Tank System environments [14] compared to a baseline PPO agent. By incorporating the environment's reward signals alongside the teacher's guidance, the approach helps align the agent's behavior with the environment. The decaying influence of the teacher facilitates a smoother transition from teacher-guided RL to independent RL.

One limitation of this approach is inconsistency in the reward structure. The agent may receive a different reward for the same state-action pair, potentially destabilizing training.

Z. Wang et al. [15] proposed an action masking technique to improve training efficiency across different environments. Unlike the reward-shaping approach [16], where the agent can select any action and learn from the resulting signal, this method directly limits the agent's action space. The teacher model outputs a binary action mask, where each bit corresponds to a possible action. This mask is applied to the agent's output, preventing it from selecting certain actions (i.e., setting their probability to 0).

This study's approach was evaluated in three environments: a simple maze environment, MinAtar (scaled down version of Atari) and μRTS2 (a simple real-time strategy game). The RL algorithms used were PPO and Deep Q-Network (DQN) - the value-based algorithm described in Chapter 2. The action-masking approach resulted in noticeably more efficient training (i.e., convergence in fewer timesteps) compared to the baseline PPO and DQN agents in all environments except for MinAtar, where performance was comparable.

One missed opportunity in this study was the lack of exploration into a gradually decaying mask rather than a strictly binary one. However, the paper does mention "the primary role of an action mask is to prevent the agent from sampling invalid actions, necessitating a binary mask" [15].

The guided approaches discussed above require a pretrained agent or specific rules scoped to the exact environment to act as the teacher. This restricts the teacher's usefulness to the environment's specific state and action space, significantly limiting transferability and scalability. Additionally, the features being fed into these agents must be manually engineered, which can be problematic in data-rich environments like cybersecurity, where vital information may be missed in the feature engineering process.

A potential solution is to use an LLM as the external knowledge source. Textual environment data can be fed directly into the LLM, alleviating concerns about missing vital information due to manual feature engineering. Further-

more, since the LLM is not trained on the specific environment, it can be applied across multiple RL environments, enhancing overall scalability and transferability.

3.2 LLMs and RL Integration

As discussed in the background, an LLM is fundamentally a Deep Neural Network (DNN) designed to recognize patterns in language [11]. The vast number of parameters and the unique transformer architecture [20] enable LLMs to recognize complex patterns in language, and generate contextually relevant responses for a wide range of Natural Language Processing (NLP) tasks.

One of the challenges of integrating an LLM into the RL pipeline is that the LLM's textual response cannot be directly executed in the environment. W. Huang et al. [53] proposed a solution to this problem using a pretrained LLM to perform actions in the VirtualHome environment [53], which simulates basic human household activities and returns a boolean indicating whether the task was completed successfully. This study demonstrated how an LLM's textual output can be mapped to an executable action using RoBERTa [54], an encoder-only type LLM. The process of this mapping is illustrated in Figure 3.1 [53].

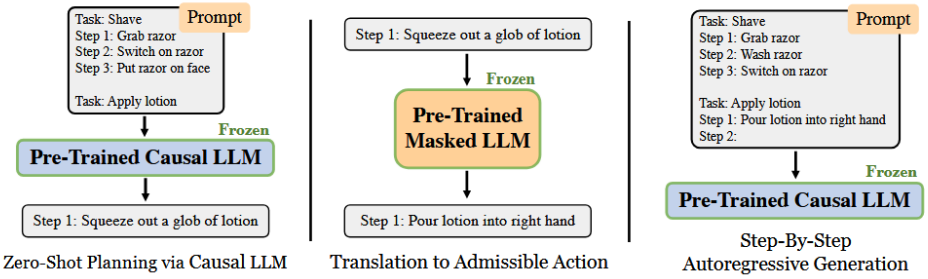


Figure 3.1: Mapping of LLM-generated text to executable actions in the VirtualHome environment [53].

One limitation with this work is that the environment is very simplistic, and the study did not conduct any testing in a more realistic, complex setting. Additionally, while the study referred to this method as zero-shot learning, it is actually few-shot learning, since examples are attached to the prompt to provide the LLM with additional context.

The study also stated that "repetitive trial-and-error is equivalent to probing the environment for privileged information, which should not be considered viable in our setting" [53]. However, this concept is the basis of RL and would have been very beneficial to explore, as improvement could occur through direct interaction with the environment. This lack of learning is the study's most significant limitation.

This deficiency is addressed in the solution proposed by M. Kwon et al. [16], which uses GPT-3 [55], a decoder-only LLM to compute rewards for the RL process. The method involves including the user's intent (i.e., the objectives) in the LLM's prompt, which is then compared to the outcome of an episode. The LLM evaluates whether the outcome of the episode aligns with the user's intent and outputs a boolean (yes or no). The RL algorithm used in this study was DQN.

One limitation of this approach is that the agent must wait for the entire episode to complete before updating its policy. Additionally, this system struggles with capturing the details of complex, dynamic environments (like those modeling cybersecurity), where actions often exist on a spectrum of favorability, which is best captured by a continuous reward signal.

Another limitation is that the objective must remain static throughout an episode, whereas, in complex environments, objectives can change dynamically. Moreover, the user must manually create a prompt for the LLM to evaluate whether the objective was met. This not only adds extra effort, but also requires the user to have knowledge on prompt engineering to effectively convey their intent.

This study's testing was limited to short-horizon games with very small step sizes. While it effectively demonstrates how an LLM can enhance reward signals, ultimately yielding a better policy, it misses the opportunity to explore how LLMs could improve RL training efficiency.

A similar approach is proposed by S. Tu et al. [17], in which the RL agent outputs two possible actions, and the LLM selects the best one. The preferred action is then used to train a separate reward model (or fine-tune an existing one), which in turn helps guide the RL process.

The study evaluated this method using the MetaWorld [17] benchmark, which consists of multiple simulated robotic manipulation tasks. The Soft Actor-Critic (SAC) [17] RL algorithm was used - an off-policy actor-critic method that combines value-based and policy-based approaches. In SAC, the actor network outputs a probability distribution across actions, while the critic assigns a value indicating how favorable the selected action is. This combination typically improves stability due to the critic's guidance, but is more complex to implement and tends to have higher variance than purely

value-based methods like DQN, as the actor's stochastic policy introduces randomness.

This study used GPT-4o-Mini [17], a decoder-only LLM to critique the RL agent's outputs. Their methodology outperformed or matched the performance of a baseline SAC agent in terms of policy convergence across most MetaWorld scenarios. These results demonstrate superior policy convergence and further highlight the benefits of LLM integration in RL environments. It should be noted that the time required to train the reward model does not appear to be included in the evaluation metrics; however, the study does not explicitly confirm this.

The biggest disadvantage of this approach is the overhead of training a reward model, which increases overall training time. However, this idea of distilling the LLM's output into a smaller reward model provides a resource-efficiency benefit - once trained, the reward model can guide training without the expensive computational costs associated with an LLM. The authors also noted that this approach works well for commercial use cases, where users may have limited access to an RL environment's underlying code.

L. Chen et al. present *RLingua* [18], which uses GPT-4 to generate an initial policy that an RL agent can leverage for robotic tasks. The policy is generated in the form of a Python script that guides the robot's actions. A one-shot learning approach is used, where an example template is included in the prompt to ensure relevance.

RLingua uses the Twin-Delayed Deep Deterministic Policy Gradient (TD3) RL algorithm [56], a hard actor-critic method that converges onto a deterministic policy - meaning that the actor outputs a single action rather than a probability distribution. The critic in TD3 is similar to DQN [44], but consists of two separate networks that each output a Q-value. The lowest Q-value is selected as the final output to mitigate the overestimation bias, a common issue in standard value-based RL algorithms [56].

To balance exploration and exploitation, RLingua uses an epsilon-greedy approach, where the agent has a probability of ϵ to select a random action. However, instead of choosing an arbitrary action, RLingua selects an action from the LLM-generated policy. The probability of sampling from the LLM's policy decays over time, gradually shifting toward exploitation as the agent becomes optimized for the environment.

Panda_Gym and RLBench [18] were the initial simulated robotic environments used for this evaluation. The study then extended testing to an emulated environment where a real robot performed tasks such as placing cubes in a bucket. The approach successfully reduced training time in both simulated and emulated environments.

RLingua effectively leverages an LLM to accelerate training; however, similar to M. Pfeiffer et al., the LLM guidance is provided in isolation from the environment. This means the LLM's advice will be independent of the observed state, potentially providing suboptimal recommendations for edge cases. This issue is particularly concerning in stochastic environments like cybersecurity, where conditions change dynamically.

Additionally, the study could have experimented with LLMs specialized in the robotics domain instead, rather than relying on GPT-4, a general-purpose model. An LLM that has been explicitly trained on robotics data could have produced more relevant and accurate recommendations.

Another limitation is the deterministic policy to which the agent converges. In complex environments like cybersecurity, multiple actions may be equally valid for a given state (e.g., isolating or restoring an infected system). A stochastic policy, where actions are sampled from a probability distribution, would allow for more efficient exploration. This approach eliminates the need to manually integrate exploration techniques like the epsilon-greedy method.

Z. Zhou et al. [19] proposed *LLM4Teach*, a method where a pretrained LLM acts as a teacher to guide an RL agent's training. The RL agent is trained using PPO [19]. A stochastic policy is outputted by the LLM, which is compared to the agent's policy to compute a "teacher loss", representing the agent's deviation from the LLM.

Specifically, the agent learns by minimizing the following loss function [19]:

$$L_{tot}(\theta) = L_A(\theta) + \lambda L_{LLM}(\theta)$$

where $L_A(\theta)$ is the original actor loss and $L_{LLM}(\theta)$ is the loss calculated by comparing the RL agent's policy to the LLM's policy. λ is decreased over time, reducing the LLM's impact on the agent's loss. Two decoder-only type LLMs are used in this study: ChatGLM-Turbo [57] and Vicuana [58].

To generate a distribution over possible actions, the LLM was queried multiple times for each action. The authors noted this inefficiency, and suggested the possibility of directly accessing the model's logits to map these to a distribution across possible actions, ultimately requiring a single forward pass.

Additionally, blending the LLM's probability distribution with the RL agent's policy could slow convergence if any misalignment is present, as the LLM directly impacts the agent's policy gradient. This presents an opportunity to explore alternative integration methods, such as action masking or feature space modification.

This study included a comprehensive evaluation, comparing the LLM-enhanced RL agent (*LLM4Teach*) against both a baseline RL agent and

the LLM's standalone performance. Their implementation was evaluated in the MiniGrid and Habitat environments [19]. MiniGrid is a simple environment where the agent learns to navigate a grid world, whereas Habitat is more complicated and involves performing robotic tasks such as navigating to and picking up an object. Their method successfully reduced training time in both environments.

L. Chen et al. [59] proposed using a transformer - the same architecture used for LLMs to process actions, states, and rewards across timesteps. This approach eliminates the need to compute the Bellman equation, which accounts for future rewards [41]. Instead of RL, the self-attention mechanism allows the transformer to leverage past experiences, encoded within the input sequence to select actions. Figure 3.2 illustrates how the transformer is used across timesteps [59].

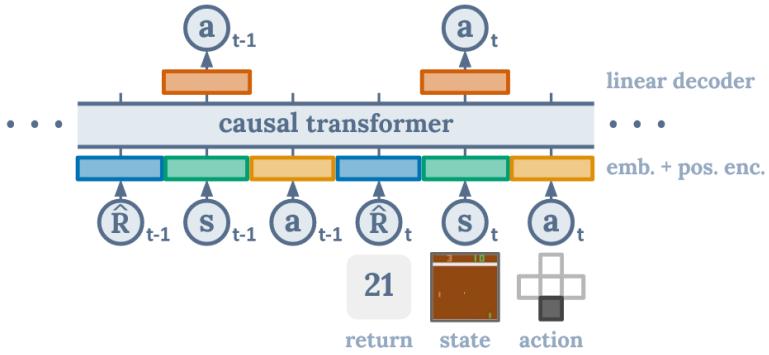


Figure 3.2: Replacing traditional RL agents with a transformer-based model for decision-making [59].

This method was evaluated against value-based, offline approaches as baselines. In simpler environments such as Key-to-Door [59], the transformer-based method outperformed traditional RL. However, in more complex environments like Atari [59], performance declined compared to baseline RL implementations. Performance was measured as final convergence rather than training efficiency in this work.

This study proposed a novel implementation; however, one major issue with this approach is the transformer's limited context length. In simple environments, the model can be fed the entire state, action, and reward sequence across an episode; however, this is rarely the case for complex environments, such as those that model the cybersecurity domain.

J. Wang et al. [60] present another novel approach for integrating LLMs into the RL pipeline. The scenario for this study requires an RL agent to

navigate a photo-realistic environment using textual instructions that describe the desired actions.

The RL agent is trained using the Advantage Actor-Critic (A2C) method [60], where it receives positive rewards for progressing to the destination and negative rewards for diverging from the destination. Since the environment provides an abundance of reward signals, this study focused on instruction comprehension rather than reward shaping. This was achieved by having the ChatGLM-6B LLM [60] break down the complex initial instructions, into shorter, more concise ones which were then fed into the agent. This idea is illustrated in Figure 3.3 [60].

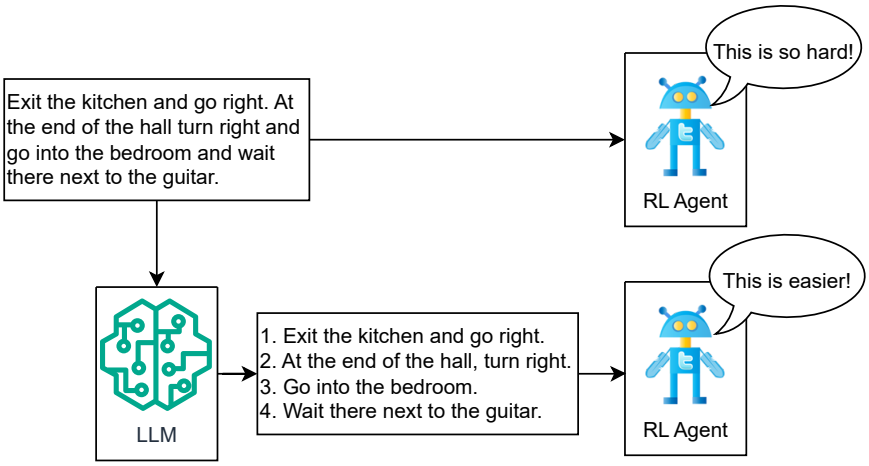


Figure 3.3: An LLM decomposing instructions into sub-tasks for improved processing by an RL agent [60].

The study demonstrates that their technique outperforms baseline PPO and A2C implementations, achieving more successful navigations while requiring fewer steps to achieve a successful outcome.

Their approach of modifying the feature space is unique, as it does not directly alter the agent's policy gradient; rather, it modifies the information available for making an effective decision. In theory, this should result in more stable learning; however, it may be slower than the above approaches as the agent must learn how to map the modified feature space to executable actions recommended by the LLM.

Furthermore, this study only integrates the LLM's guidance into the critic's feature space - the actor's feature space is produced solely from the environ-

ment's visual and orientation information. An interesting extension would be to incorporate the LLM guidance into the actor's feature space, allowing the agent to directly learn actions corresponding to the LLM's instructions. However, this could also introduce a risk of over-reliance on the LLM.

Another unexplored approach in this study is having the LLM augment the existing information provided by the environment. Instead of simply being fed raw navigation instructions, the LLM could be fed additional environment-specific details to help it generate more contextually relevant responses. This existing information could include the agent's recent actions and its proximity to objects, enabling the LLM to create more tailored instructions to the agent's current state.

Previous work on LLM-RL integration has primarily focused on overcoming sparse reward signals in their respective environments. However, in cybersecurity, the frequent feedback signals present an opportunity to focus on how LLMs can directly augment the agent's decision-making capabilities rather than being used to compensate for environmental limitations.

Furthermore, most of these studies have used generic LLMs, trained on broad, multi-domain knowledge. This presents an opportunity to experiment with LLMs that have been pretrained in a particular domain (i.e., an LLM pretrained in the cybersecurity domain).

Finally, decoder-only LLMs were the predominant focus in these studies. Experimenting with encoder-only models where the input is processed bidirectionally could be beneficial. This is particularly relevant in network security, where the order of hosts does not hold particular importance. For example, if the hosts for *User1* and *User2* are compromised, their position in the prompt should not influence the LLM's decision.

3.3 LLMs in Cybersecurity

The LLM integration discussed in the previous section focused on RL, but was not directly related to cybersecurity. This section explores how LLMs can be leveraged for cybersecurity applications, particularly threat detection.

M. Guastalla et al. [61] demonstrate how LLMs can be leveraged to detect Distributed Denial of Service (DDoS) attacks, one of the most common cyber threats. This study explores prompt engineering techniques (specifically few-shot learning) on baseline GPT-3.5 and GPT-4 models, as well as a fine-tuned Ada model [61]. The Ada model is another decoder-only type model with significantly fewer parameters than the GPT-based models. The Ada model

was fine-tuned on the CICIDS 2017 and Urban IoT datasets [61], which contain classified DDoS attack samples.

To evaluate performance, a baseline Multilayer Perceptron (MLP) shown in Figure 3.4 was trained using the same datasets [61].

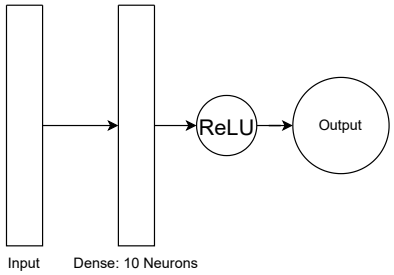


Figure 3.4: Basic Multilayer Perceptron (MLP) used for performance evaluation [61].

Findings show that both the pretrained and fine-tuned LLMs outperform the baseline MLP in DDoS detection. However, a limitation of this study was the choice of a very basic MLP as the baseline. A more advanced MLP with multiple dense layers and more parameters could have provided a more realistic benchmark for comparison.

Tarek Ali and Panos Kostakos' *HuntGPT* [62] provides another example of how LLMs can be applied to cybersecurity. Unlike previous studies where the LLMs acted as black box models, HuntGPT aims to increase transparency by providing explanations for why a specific sample is deemed malicious and what mitigation measures can be employed.

A random forest classifier trained on the KDD99 dataset [62] is used to detect anomalies, while HuntGPT provides explanations for why a specific sample is deemed malicious and recommends countermeasures. The KDD99 dataset is a popular benchmark dataset containing labeled data for intrusion detections and general network security.

HuntGPT's baseline model is ChatGPT-3.5 [62]. Instead of fine-tuning the model, the study employs Retrieval-Augmented Generation (RAG) to tailor the LLM's response to cybersecurity.

This study demonstrates that HuntGPT can effectively and accurately provide insights into cybersecurity-related actions. However, a missed opportunity was integrating HuntGPT into the detection process itself, which could have potentially increased performance compared to their random forest implementation.

J. Loevenich et al. [63] proposed leveraging external knowledge by integrating an LLM into the CybORG [2] environment. However, similar to Tarek Ali and Panos Kostakos' HuntGPT [62], this approach was used to increase transparency rather than impact decision-making. As discussed in Section 2.3, CybORG is a cybersecurity environment designed for ACO.

The study used ChatGPT-3.5 [63] as the baseline model, without any fine-tuning. Similar to HuntGPT [62], RAG was used to transparently augment prompts with knowledge from MITRE'S ATT&CK framework [63]. The findings reinforce that LLMs can recognize patterns in cybersecurity data, highlighting their potential benefit in improving training efficiency for ACO.

These examples illustrate how LLMs can be applied to cybersecurity; however, they rely on the LLM's static configuration to make (or explain) decisions. This lack of adaptability is not ideal in the dynamic, ever-evolving cyber environment, where new threats continuously emerge.

Integrating RL with LLMs offers significant advantages for cybersecurity. It mitigates the concerns about novel threats that may not be present in the potentially outdated data used during LLM training, allowing these systems to dynamically adapt to emerging threats in a scalable manner. Furthermore, these studies highlight how LLMs can increase transparency in decision-making, a crucial aspect in cybersecurity, where the impact of actions can be immense.

3.4 Autonomous Cyber Operations

RL is the main technique employed for modern ACO applications, which offers an alternative solution to signature-based approaches by allowing ACO agents to learn through direct interaction with the environment. Unlike the Machine Learning (ML) approach proposed by A. Bhagalaskmi et al. [64], RL eliminates the need to manually create datasets, making it effective for ACO. As discussed in Section 2.3, CybORG [2] is a cybersecurity environment designed to train both blue (defensive) and red (offensive) RL agents. Previous environments have been created to simulate the cyber domain, but many were either not scalable or not designed for RL, where sufficient signals are required to guide training. Table 3.1 outlines key differences between various cybersecurity environments [2].

Table 3.1: Comparison of cybersecurity environments [2].

Environment	**Scalable**	**Flexible**	**Efficient**	**Designed for RL**
DETERlab	Low	Yes	Low	No
VINE	Med	Yes	Med	No
SmallWorld	Med	Yes	Med	No
BRAWL	Med	Windows	Med	Limited
Galaxy	Low	Debian-based	High	Yes
Insight	High	Yes	Med	Yes
CANDLES	High	Yes	High	Yes
Pentesting Simulations	High	Yes	Med	Yes
CyAMS	High	Yes	High	Yes
CybORG	**High**	Yes	**High**	**Yes**

The most significant limitation of the original CybORG environment is its sparse reward system, where the agent only receives a win or loss signal at the end of an episode. This is not representative of real-world cybersecurity, where rewards should be obtained throughout episodes. For example, the red agent should be rewarded for intermediate steps such as establishing an initial foothold on the network, not just for achieving the ultimate target.

This issue was addressed in CybORG's Cage Challenge 2 [47], where rewards are received after each timestep, allowing for more stable RL training.

While significant work has leveraged CybORG to develop ACO [7, 6, 65], existing approaches require agents to learn optimal policies from scratch. Incorporating external cybersecurity knowledge could reduce training time and potentially lead to more effective policies.

Furthermore, it should be noted that Cage Challenge 2 [50] still has limitations, including a lack of realistic actions for the blue agent, a state representation that does not fully capture the complexities of an enterprise network, and simplistic preprocessing of CybORG's state into a numerical feature space for RL agents. The open-source nature of Cage Challenge 2 presents a great opportunity to optimize these aspects.

3.5 Artificial General Intelligence

This section discusses Artificial General Intelligence (AGI) and argues that its emergence will not render research in the described fields obsolete.

The concept of AGI was first introduced by Ben Goertzel in 2006 [66] and is anticipated to emerge by 2029 [67]. S. Bubeck et al. define AGI as "a brilliant oracle, for example, that has no agency or preferences, but can provide accurate and useful information on any topic or domain" [68].

The idea behind AGI is to create an agent that does not just generate responses by altering inputs through a series of weights, but is able to reason like a human and generate responses on domains it has not been explicitly trained on. It should be noted that popular LLMs like ChatGPT are still considered Artificial Narrow Intelligence (ANI) due to their limitations in areas such as confidence calibration, long-term memory, personalization, reasoning and planning [68].

The current leading approach to achieving AGI is through integrated cognitive architectures [69], which combine neural networks with other cognitive systems to simulate the human thought process.

At first glance, one might assume that an AGI agent, capable of reasoning across all domains would outperform any method produced by this research. However, several fundamental limitations of AGI ensure research in these fields remains relevant, even if AGI emerges in the near future.

3.5.1 Generalized Nature

Due to its generalized nature, AGI would struggle in specialized environments such as cybersecurity. Tailoring an AGI agent to produce domain-specific responses would require extensive fine-tuning and computational resources, reducing the effectiveness of existing fine-tuning techniques [31].

This challenge is particularly critical in cybersecurity, where new and emerging threats evolve rapidly. An AGI agent may struggle to adapt quickly enough without continuous retraining, limiting its real-world adaptability.

3.5.2 Computational Requirements

The computational resources required to support AGI are immense, estimated at 10^{16} computations per second [70]. For comparison, this is approximately 2.5 **million** times more powerful than a 4 GHz processor, which represents the higher end of commercially available processors at the time of writing.

3.5.3 Transparency

Due to its underlying complexities, AGI would likely suffer from a lack of explainability, making it a black-box system with limited transparency regarding how it arrives at decisions. In particular, this poses issues in cybersecurity, where auditing and debugging are essential. Without clear insights into the decision-making process, it would be difficult to trust its assessments, potentially limiting its adoption in critical systems like cybersecurity.

3.5.4 Ethical Concerns

Beyond the transparency issues, ethical considerations could hinder the adoption of AGI. One major concern is AGI's potential impact on the global workforce [67]. Automation at an unprecedented scale could lead to job displacement, prompting governments to implement regulations that may limit or even halt the deployment of AGI.

The technical and ethical challenges outlined above demonstrate that methodologies developed in this research will remain relevant, even in the presence of AGI.

3.6 Discussion

There has been substantial progress in integrating RL into ACO [64, 2, 47, 7, 6, 65]. However, current approaches require agents to learn from scratch, ultimately leading to longer training times and the need to perform suboptimal actions to learn their consequences - a critical limitation in dynamic, adversarial environments. Prior studies [13, 14, 15] have demonstrated that guiding the RL process using a teacher model can increase training efficiency, but these techniques have yet to be applied within the ACO domain.

Existing research [61, 62, 63] has shown that LLMs can recognize patterns in data pertaining to cybersecurity and generate contextually relevant responses. This makes LLMs a strong candidate for the teacher role in the RL process for ACO.

Despite the promising overlap of LLMs, RL, and ACO, limited research has been conducted in this area. J. Lovenich et al. [63] integrated an LLM into the CybORG environment [2], but this was used to increase transparency between the end-user and the RL agent's decisions. The LLM had no impact on the training process.

3.6.1 Research Opportunities

These unresolved issues present a valuable opportunity to explore LLM integration in RL for ACO, with the primary goal of augmenting decision-making.

Furthermore, the discussed teacher-guided techniques have mostly been studied in isolation of each other, presenting an opportunity to combine methodologies (e.g., using auxiliary loss alongside action masking).

4 Methodology

This chapter outlines the phased approach used to integrate a Large Language Model (LLM) into the Reinforcement Learning (RL) process to augment decision-making in Autonomous Cyber Operations (ACO).

These phases correspond to those described in Section 1.4 and include:

- **Selecting an LLM**. The LLMs were systematically reviewed using a dataset of predefined CybORG-specific questions, enabling the selection of the most suitable model.
- **Environment Modification**. CybORG's Cage Challenge 2 [50] environment was modified to better reflect realistic cybersecurity conditions and facilitate more efficient agent development. This included:
  - Altering the action space.
  - Updating the existing wrapper functionality.
  - Modifying the RL agent's feature space.
- **Baseline Agent Development**. Baseline agents were developed for direct comparison against the LLM-guided ones. This was achieved by developing Deep Q-Network (DQN) and Proximal Policy Optimization (PPO) agents and comparing their performance.
- **Teacher-Guided Algorithm Development**. Various teacher-guided algorithms were developed and compared using the ideas and techniques discussed in Section 3.1.
- **LLM Integration into the RL Pipeline**. The best-performing LLM from Phase 1 was incorporated into the existing RL pipeline to augment decision-making. This involved:
  - Refining prompt engineering beyond the techniques developed in Phase 1 for the selected LLM.
  - Implementing a method to reliably extract an executable action from the LLM.
  - Integrating the LLM into the decision-making process using the best-performing teacher-guided technique.

4.1 Selecting an LLM

To identify an LLM capable of generating responses that are contextually relevant to CybORG, six open-source LLMs pretrained on cybersecurity-related data were selected for evaluation. These were chosen based on open-source availability, baseline architecture, and existing reviews. The evaluation process involved validating the performance of these LLMs on a dataset of cybersecurity-related question-answer pairs. Table 4.1 shows the LLMs involved in the evaluation process.

Table 4.1: LLMs involved in the evaluation process.

LLM Name	**Baseline Architecture**	**LLM Source**
CyberBase [8]	Vicuana-13B	Hugging Face
Cyberdost2b [9]	Navarasa-2.0-2B	Hugging Face
HackMentor [71]	Llama-13B	GitHub
Lily7B [10]	Mistral-7B	Hugging Face
Cyber-Risk-Llama8B [72]	Meta-Llama-8B	Hugging Face
Z7sec [73]	Zephyr-7B	Hugging Face

To ensure the LLMs were provided with input that could be effectively analyzed to generate a relevant response, initial prompt engineering was conducted based on existing best practices [74, 34]. This evaluation focused on the LLM's baseline knowledge, requiring prompts to be generic rather than task-specific. For example, the definition of an action in the context of CybORG was not included in the prompt - the evaluation was to determine the capabilities of the LLMs' existing training. To ensure an unbiased assessment, the prompt format was identical for each evaluated LLM and is illustrated in Figure 4.1.

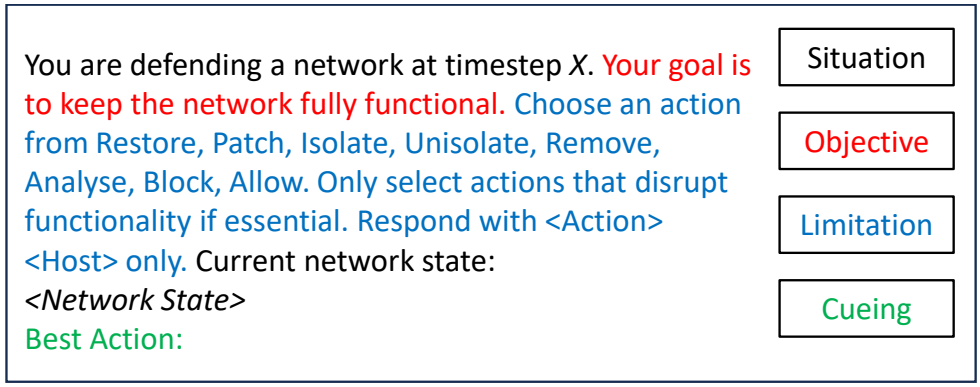


Figure 4.1: Initial prompt design for evaluating LLMs. For clarity, the components of the prompt are color-coded.

Initial testing showed that including the raw CybORG output directly into the prompt (denoted by *Network State* in Figure 4.1) yielded poor responses - this will be discussed further in the subsequent chapter. To augment the LLM's ability to extract meaningful patterns from the environment, CybORG's state was parsed into a condensed JavaScript Object Notation (JSON) form and a condensed sentence form, both of which were included in each question for the evaluation dataset. These formats are shown in Figure 4.2.

Raw CybORG State

"Enterprise0": {"Processes": [{"Connections": [{"local_port": 22, "remote_port": 57474, "local_address": "10.0.179.156", "remote_address": "10.0.104.214"}]}], "Interface": [{"IP Address": "10.0.179.156"}], "System info": {"Hostname": "Enterprise0", "OSType": "LINUX", "OSDistribution": "UBUNTU", "OSVersion": "U18_04_3", "Architecture": "x64"}, "Isolated": False},

"Enterprise1": {"Processes": [{"Connections": [{"local_port": 22, "remote_port": 53406, "local_address": "10.0.179.147", "remote_address": "10.0.104.214"}]}, {"Connections": [{"local_port": 135, "remote_port": 52580, "local_address": "10.0.179.147", "remote_address": "10.0.104.214"}]}, {"Connections": [{"local_port": 3389, "remote_port": 50435, "local_address": "10.0.179.147", "remote_address": "10.0.104.214"}]}, {"Connections": [{"local_port": 445, "remote_port": 51786, "local_address": "10.0.179.147", "remote_address": "10.0.104.214"}]}, {"Connections": [{"local_port": 139, "remote_port": 58258, "local_address": "10.0.179.147", "remote_address": "10.0.104.214"}]}, {"Connections": [{"local_port": 80, "remote_port": 49897, "local_address": "10.0.179.147", "remote_address": "10.0.104.214"}]}, {"Connections": [{"local_port": 443, "remote_port": 59506, "local_address": "10.0.179.147", "remote_address": "10.0.104.214"}]}], "Interface": [{"IP Address": "10.0.179.147"}], "System info": {"Hostname": "Enterprise1", "OSType": "WINDOWS", "OSDistribution": "WINDOWS_SVR_2008", "OSVersion": "W6_1_7601", "Architecture": "x64"}, "Isolated": False}

Condensed JSON State

"Enterprise0": {"ip": "10.0.179.156", "Processes": [{"remoteAddress": "10.0.104.214", "localPort": 22, "count": 1}], "Files": [], "Isolated": False, "LastAnalysed": -1, "Priority": "Medium"},

"Enterprise1": {"ip": "10.0.179.147", "Processes": [{"remoteAddress": "10.0.104.214", "localPort": 22, "count": 1}, {"remoteAddress": "10.0.104.214", "localPort": 135, "count": 1}, {"remoteAddress": "10.0.104.214", "localPort": 3389, "count": 1}, {"remoteAddress": "10.0.104.214", "localPort": 445, "count": 1}, {"remoteAddress": "10.0.104.214", "localPort": 139, "count": 1}, {"remoteAddress": "10.0.104.214", "localPort": 80, "count": 1}, {"remoteAddress": "10.0.104.214", "localPort": 443, "count": 1}], "Files": [], "Isolated": False, "LastAnalysed": -1, "Priority": "Medium"}

Condensed Sentence State

Enterprise0 | IP: 10.0.179.156, Priority: Medium, Isolated: No, Last Analysed: -1, Processes: [1 process with: (Remote IP: 10.0.104.214 and Port: 22)]

Enterprise1 | IP: 10.0.179.147, Priority: Medium, Isolated: No, Last Analysed: -1, Processes: [1 process with: (Remote IP: 10.0.104.214 and Port: 22), 1 process with: (Remote IP: 10.0.104.214 and Port: 135), 1 process with: (Remote IP: 10.0.104.214 and Port: 3389), 1 process with: (Remote IP: 10.0.104.214 and Port: 445), 1 process with: (Remote IP: 10.0.104.214 and Port: 139), 1 process with: (Remote IP: 10.0.104.214 and Port: 80), 1 process with: (Remote IP: 10.0.104.214 and Port: 443)]

Figure 4.2: Illustration of the same CybORG scenario parsed in three different ways. The first is the raw output of the CybORG environment. The second and third are the corresponding condensed JavaScript Object Notation (JSON) and sentence forms, respectively. For clarity, only 2 of the 13 hosts' states are shown.

Questions of varying difficulty, following the prompt structure of Figure 4.1 were created to evaluate the selected LLMs. The difficulty was determined by the number of hosts involved:

- Easy: 1-2 hosts.
- Medium: 3-7 hosts.
- Hard: 8-13 hosts.

A Python script was developed to generate these questions, incorporating stochasticity to cover a variety of scenarios while implementing predefined rules to ensure the constraints of the CybORG environment were respected. For example, a scenario would not have a compromised operational server at timestep 0. These scenarios were converted to both the JSON and sentence formats denoted in Figure 4.2, and parsed into corresponding questions using the generic prompt structure described in Figure 4.1. A total of 100 questions were created, consisting of 20 easy, 40 medium, and 40 hard.

Each question was assigned a corresponding answer by manually selecting the most contextually relevant action in CybORG's predefined action-space. Emphasis was placed on selecting the action that would yield the highest reward for the episode, not necessarily the individual timestep.

For evaluating the LLMs, a script was developed to compare their responses against the manually selected answer. This comparison utilized BERTScore

[12], an encoder-only type LLM that computes a relevance score between two pieces of text using cosine similarity. The relevance score consists of a precision, recall, and F1 (metric that balances precision and recall) component.

Manually assigned scores were chosen as the primary evaluation criterion; however, precision was prioritized over recall for BERTScore's output. This is because precision is inversely proportional to the number of false positives - meaning that to achieve a high precision score, the LLM must minimize the number of irrelevant tokens. This is contrary to recall, which is inversely proportional to the number of false negatives. A high recall score requires the LLM to maximize the number of correct tokens, even if it includes irrelevant ones. For example, if the ground truth label is: '*Isolate Host A*', but the LLM outputs '*Isolate, Patch, Restore, Remove Host A Host B Host C*', the recall would be high because the LLM captured all relevant tokens; however, the precision would be low since it generated many unrelated tokens. Encouraging this behavior is problematic, since it prevents the extraction of a single, concrete action from the LLM's output, making it inefficient for decision-making.

To facilitate manual validation, the LLM's textual output for the JSON-based and sentence-based questions was recorded in addition to BERTScore's metrics. The complete process of selecting an LLM is illustrated in Figure 4.3.

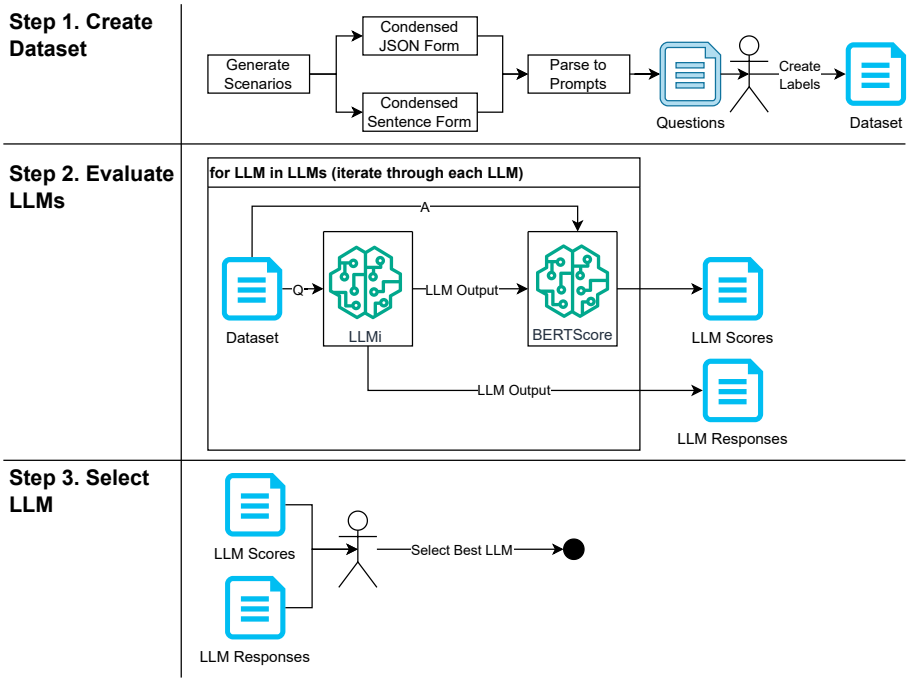


Figure 4.3: Process used for selecting an LLM. In Step 1, a dataset of questions and corresponding answers was created to evaluate the LLMs. In Step 2, each of these LLMs was evaluated against this dataset, using BERTScore to compute a precision, accuracy, and F1-score for each prediction. In Step 3, the results from Step 2 were manually evaluated and the best-performing LLM was selected.

4.2 Environment Modification

To better reflect realistic cybersecurity conditions, certain actions were added to the Cage Challenge 2 environment used for this research [50]. The actions include:

- Block. Prevents communication between two hosts.
- Patch. Decreases the chance of an attacker's exploit succeeding on a host.
- Isolate. Simulates unplugging the host from the network - no connectivity to or from any host.

4.2.1 Block Action

The block action was derived from Cage Challenge 4 [52], one of CybORG's environments, which is designed for multi-agent development. The action was implemented to enable the blue agent to deny new connections between two hosts, simulating typical Intrusion Prevention System (IPS) [1] functionality. This action does not disrupt existing sessions, and only applies to new connections originating from the source host to the destination host.

Additionally, to ensure a consistent action space - where the same set of actions is available across all hosts and timesteps, the block action is applied across all ports. Since each host runs different services, the available ports to block would vary between hosts, unless every possible port was included as its own option. Alternatively, ports not available on a specific host could be masked (i.e., made impossible to select) during training; however, blocking individual ports would be too slow in the context of CybORG, rendering this action ineffective compared to others. The corresponding iptables rule, one of Linux's utilities for managing firewalls, for "Block HostA, HostB" is described as:

```
iptables -A INPUT -s HostA -d HostB -m state --state NEW -j DROP
```

Blocked hosts are tracked using a Python dictionary, where each key represents the source host, and each value is a list of blocked destination hosts. When the block action is executed, the corresponding source (first argument) and destination (second argument) hosts are added to the dictionary. Block is the only action that takes two hosts as an argument.

To enforce blocking, the behavior of certain red actions was modified:

- DiscoverNetworkServices (discover services on a host) and ExploitRemoteService (gaining foothold on host) were modified to not execute if the source and destination hosts are present in the block list.
- DiscoverRemoteSystems (discover hosts on a subnet) was modified to not return any hosts present in the block list.
- PrivilegeEscalate (escalating to admin-level privileges) and Impact (attacker's end goal - e.g., data exfiltration) remained unaffected because they typically leverage existing sessions rather than create new ones.

A negative reward was implemented for each blocked host to discourage the blue agent from simply blocking connectivity from the beginning, which would significantly degrade network functionality. This negative reward is applied consistently at every timestep that the block is in place. The penalty is proportional to the importance of the blocked hosts, ensuring that low-value

disruptions are interpreted differently than high-value ones. For example, blocking connectivity from a user host to an enterprise host yields a lower penalty than blocking connectivity between an enterprise host and the operational server. The corresponding rewards for block with respect to the destination host's priority are:

- -0.1 for user hosts; and
- -0.2 for enterprise servers and operational hosts; and
- -0.3 for the operational server.

An overview of the block action is depicted in Figure 4.4.

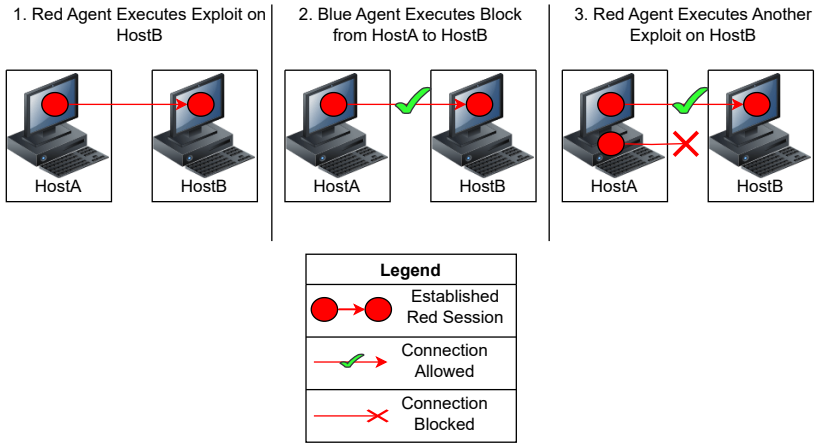


Figure 4.4: Illustrating the functionality of the block action. The red agent propagates from HostA to HostB (left); the blue agent blocks communications from HostA to HostB (middle), the red agent's existing session is uninterrupted, but cannot initiate new connections to HostB (right).

An unblock action was also implemented that takes two hosts as parameters. Rather than adding hosts to the block list, it removes them, effectively re-enabling connectivity between the source and destination. At the start of an episode, no host is in the block list; the only rules enforced are from the environment's configuration file.

4.2.2 Patch Action

The patch action was implemented to simulate updating vulnerable services on a host, effectively decreasing the red agent's ability to exploit a service. Similar to the block action, patch is scoped at the host level rather than targetting individual services. This design ensures a consistent action space across

hosts and maintains relevance within CybORG by ensuring that patch has a comparable impact to the other actions. If patch were applied to individual services, its scope would be smaller than that of the other host-based actions, potentially making it less desirable.

Patching is tracked by assigning a patch score to each host; the higher the score, the less likely the host is to be exploited. It was implemented as a one-dimensional list of floats between 0 and 1, where the index corresponds to the host. When the patch action is executed, the patch score for that host is increased, simulating updating/upgrading its services. At the start of an episode, each host's patch score is initialized to 0.

To enforce the effectiveness of patching, the red agent's PrivilegeEscalate and ExploitRemoteService were modified to have an X percent chance of failing where X corresponds to the host's patch score. This reflects the fact that these actions typically rely on exploiting vulnerabilities in services, which may not always be present. This randomness also helps mimic the stochastic nature of cybersecurity.

To simulate reconnaissance and prevent the blue agent from repeatedly exploiting the patch action, the red ExploitRemoteService and PrivilegeEscalate actions decrease the patch score of the corresponding host (i.e., the host the action is being performed on). Furthermore, the rate at which the red agent decreases the patch score is higher than the rate at which the blue agent can increase it. This is to prevent early overuse of the patch action and maintain its effectiveness relative to other possible actions. The patch score is decreased regardless of whether the attack succeeds. Specifically, red actions decrease the score by 35%, whereas the patch action increases it by 30%. An overview of the patch action's functionality is illustrated in Figure 4.5.

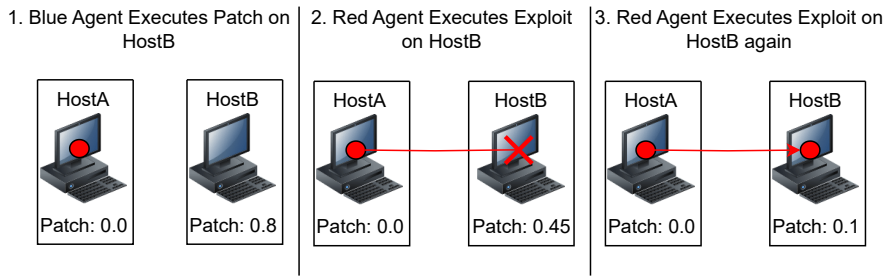


Figure 4.5: Diagram illustrating the functionality of the patch action. The blue agent patches HostB (left), increasing its patch score. The red agent attempts to exploit HostB (middle), failing the exploit, but decreasing the patch score. The red agent attempts to exploit HostB again, this time successfully establishing a session.

4.2.3 Isolate Action

The isolate action was implemented to simulate unplugging a host from the network, preventing any connectivity to or from that host. Unlike the block action, isolate disrupts existing sessions, rendering future attacks originating from or targeting that host ineffective.

Isolated hosts are tracked using a one-dimensional list of bits (1 for isolated, 0 for not isolated), where the index of each bit corresponds to the host. When the isolate action is executed, the bit for the corresponding host is set to 1. At the start of an episode, each host has an isolated value of 0, signifying full connectivity to the network.

The red agent's actions were modified to enforce isolate's functionality. In particular, DiscoverNetworkServices, ExploitRemoteService, PrivilegeEscalate and Impact are set to fail if the source (where the red session is) or destination host is isolated. DiscoverRemoteSystems (scanning a subnet for hosts) was modified to return nothing for isolated hosts and to return a failure if the host performing the scan is isolated.

To discourage the blue agent from significantly impeding network functionality, a negative reward is applied every timestep for each isolated host. This negative reward is proportional to the value of the host:

- -0.2 for user hosts; and
- -0.4 for enterprise servers and operational hosts; and
- -0.5 for the operational server.

Furthermore, the isolate action does not eliminate any of the red agent's existing sessions, it just prevents them from being used for future actions. This is to simulate the attacker establishing persistence on the machine. Isolating a host, does not remove any artifacts placed by the attacker, they must be removed via the remove or restore action. While technically unrealistic (since the attacker should not be able to access the host at all), it simulates the required functionality - preventing the red agent's use of an isolated box to further its attack.

An unisolate action was also implemented that sets the host's bit in the isolate list to a 0, simulating reconnecting the host back to the network. This action does not remove any artifacts on the host, any modifications by the red agent remain. Figure 4.6 illustrates the functionality of the isolate and unisolate actions.

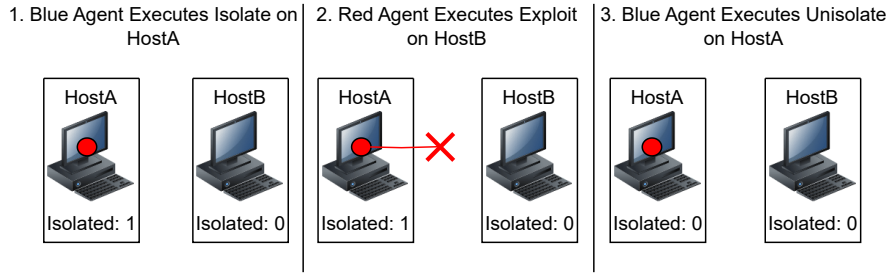


Figure 4.6: Illustration of the isolate action's functionality. The blue agent isolates HostA (left), disconnecting it from the network. The red agents attempts to exploit HostB from HostA (middle), but cannot connect since it is isolated. The blue agent unisolates HostA (right), but the red agent's session remains, as it was not explicitly removed.

4.2.4 Action Removal

The original Cage Challenge 2 environment includes six-deception based attacks, each of which sets up its own decoy service (e.g., Apache, Tomcat) on the specified host (honeypots). These actions were removed from this research because, while relevant in cybersecurity, they are not directly relevant from an ACO perspective.

In practice, honeypots are used by intelligence teams to gain information on adversaries' behaviors and techniques to facilitate the design of more robust future defenses [75]. However, the establishment of these decoy services does not provide any immediate value for defending a network compared to actions considered more proactive, like isolate and remove. To be used effectively, the network of these honeypots (i.e., a honeynet) should be pre-established as a dedicated environment for training these agents, rather than included as available actions.

Another action that was removed from this implementation was the monitor action. The reason for this is that the monitor action is automatically called at the end of every timestep to provide the blue agent with information on any new connections and processes. Because it is already called every timestep, it was removed to avoid redundancy.

Altogether, the action-space used for this research consists of seven distinct actions:

- **Analyse:** to collect further information on the host, specifically any new files on the host not contained in the baseline image.
- **Remove:** to destroy user-level malicious processes and services running on the host without disrupting network functionality.

- **Restore:** to revert the host back to its original state at the start of the episode, while disrupting network functionality.
- **Block:** to deny new connectivity between two hosts.
- **Isolate:** to unplug the host from the network, preventing the adversary from moving laterally in the network.
- **Unisolate:** to rejoin a previously isolated host to the network.
- **Patch:** to update the services running on a host, decreasing the chance of success for future adversary exploits.

4.2.5 Preprocessing

The preprocessing techniques used for this implementation build upon the default ones discussed in Chapter 2. The original CybORG environment converts the textual state of each host into an encoded representation of 4 bits, where the first 2 bits represent its activity state (i.e., the red agent's activity on the host) and the last 2 bits represent its compromised state (i.e., the red agent's presence on the host).

One modification in this implementation is that the activity and compromised states were converted into single floats, rather than pairs of bits. An advantage of using bits is that the RL agent will process the features as their own distinct categories without inherently prioritizing any of them based on their value; however, this comes at the cost of a larger feature space and the agent needing to learn which states map to which bit representation.

Because the preferred functionality is to typically prioritize more severely compromised hosts over less severely compromised ones, representing these states as a float is acceptable, because the agent will naturally treat certain categories more seriously due to their increased impact on the gradient during backpropagation. For example, if an attacker having root-level access on a host is represented by a 1.0 and user-level access is represented by 0.66, the blue agent will implicitly prioritize the root-level access, as its value contributes more strongly to the agent's training and the resulting action selection.

To enable the RL agent to utilize the patch and isolate states for action selection, these were appended as individual features for each host. The isolate state was appended as a 1 or a 0, where 1 means the host is isolated and 0 means the host is not isolated. Since patch is already a float between 0 and 1, representing how likely an exploit is to succeed, it was appended directly to the feature space.

Finally, two global features were appended: the total number of isolated hosts and the total number of compromised hosts. The number of compromised hosts includes both user-level and root-level compromises. To ensure that

these are not inherently prioritized over the other features, both of these were normalized by the total number of hosts:

$$totalIsolated_{normalized} = \frac{totalIsolated}{totalHosts}$$

and

$$totalCompromised_{normalized} = \frac{totalCompromised}{totalHosts}$$

Figure 4.7 shows the updated feature space mapping employed in this study.

Activity Mapping	
None	0.0
Scan	0.3
Exploit	1.0

Compromised Mapping	
No	0.0
Unknown	0.3
User	0.6
Privileged	1.0

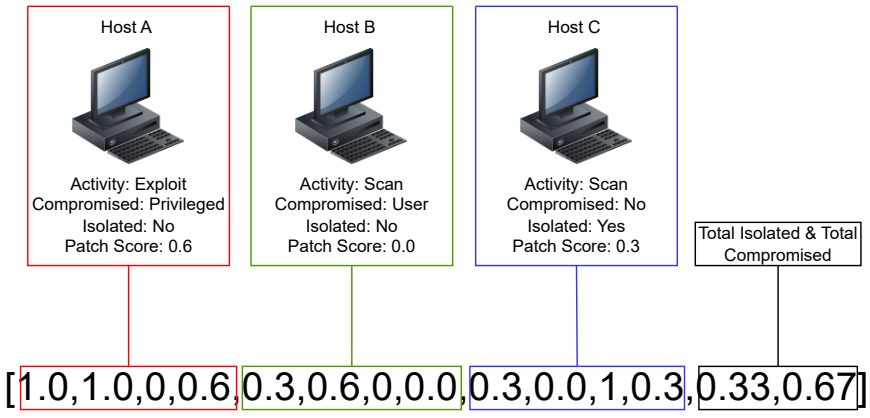


Figure 4.7: Modified feature space mapping used for this research.

4.3 Baseline Agent Development

As discussed in Chapter 2, there are two fundamental types of RL algorithms: value-based and policy-based approaches. For value-based, the agent outputs an

expected value for every possible action in a given state. Additional techniques are required to incorporate exploration, ensuring the agent explores sufficient possibilities.

For policy-based, the agent outputs a probability distribution across possible actions and samples directly from this distribution. This direct sampling eliminates the need for additional exploration techniques, since distributions are stochastic by nature.

Baseline value-based and policy-based RL algorithms were developed to evaluate the performance of the LLM integration. The intention behind this was to ensure that sufficient experimentation was conducted to identify the best baseline to compare against. In particular, Deep Q-Network (DQN) was used for the value-based agent, and Proximal Policy Optimization (PPO) was used for the policy-based agent. One of the reasons behind developing, rather than simply calling existing libraries, was to ensure the agent was implemented in such a way that a teacher could be integrated, and to gain a greater understanding of the underlying details for optimizing the LLM integration.

Ultimately, PPO was selected as the chosen RL algorithm due to its stability in training and faster convergence time. The details of these evaluations are discussed in the following chapter.

4.3.1 DQN Agent

Previous work [6, 7, 50] has primarily used PPO for the CybORG environment; however, DQN was also explored in this study due to the environment's discrete action and state space. DQN typically performs well on these discrete spaces, because they are easily represented in a table format of state-action pairs (i.e., a Q-table). For continuous spaces, there could be infinite possible Q-values making this approach less effective.

The DQN agent was based heavily on existing implementations [76, 77, 44] and is one of the most notable off-policy, value-based RL algorithms at the time of writing. As discussed in Section 2.2, DQN uses a Multi-Layer Perceptron (MLP) in place of the Q-table.

The MLP's input space consists of the numerical feature-space representing each host's state, while its output is an array of values indicating how favorable each action is for the given state (Q-values). The MLP contains three hidden layers with 256, 128 and 64 neurons, respectively. The decrementing choice was implemented because, typically in RL, only a subset of features are relevant for action selection. This ensures that the network does not overfit to features that may be irrelevant to the decision. For example, if Host A is compromised,

the network should typically prioritize the features for Host A rather than overfitting to Host B's and C's features. Having this decreasing structure, enables the model to filter out irrelevant information, leading to a more generalizable network.

Epsilon-greedy is employed to balance exploration and exploitation, where the RL agent has an ϵ chance of selecting a random action rather than the one with the highest Q-value. The value of epsilon decays as the game progresses, increasing the chances of selecting the highest Q-value as the model becomes more tuned to the environment.

A buffer was implemented that keeps track of the states, rewards, actions and next states. The size of this buffer is larger than the length of the training interval, allowing the agent to leverage data produced by previous policies in training. This is ultimately what makes this an off-policy algorithm.

For training, a separate target MLP with an identical architecture to the main one is used. This target MLP computes Q-values that are used for the loss calculation. The target Q-value for the sampled action is calculated by incorporating Temporal Difference (TD), where it is computed using the immediate reward and the target MLP's prediction for the next state, multiplied by a discount factor. In particular, the target Q-value is calculated with [41]:

$$Q_{target}(s, a) = r + \gamma * maxQ_{\theta}(s', a')$$

Where:

- s, a, r, s', a' are the current state, current action, current reward, next state and next action respectively.
- γ is the discount factor used to prioritize immediate rewards over future rewards.
- The maximum Q-value is chosen from the target network to get an estimate of the next state's value.

Once the corresponding label is calculated, Mean Square Error (MSE) is used to calculate the loss between the sample and the prediction. This loss is then backpropagated to adjust the agent's parameters. To stabilize training, the target MLP that is used to calculate the labels is updated less frequently than the main MLP.

Figure 4.8 shows the DQN algorithm that was implemented.

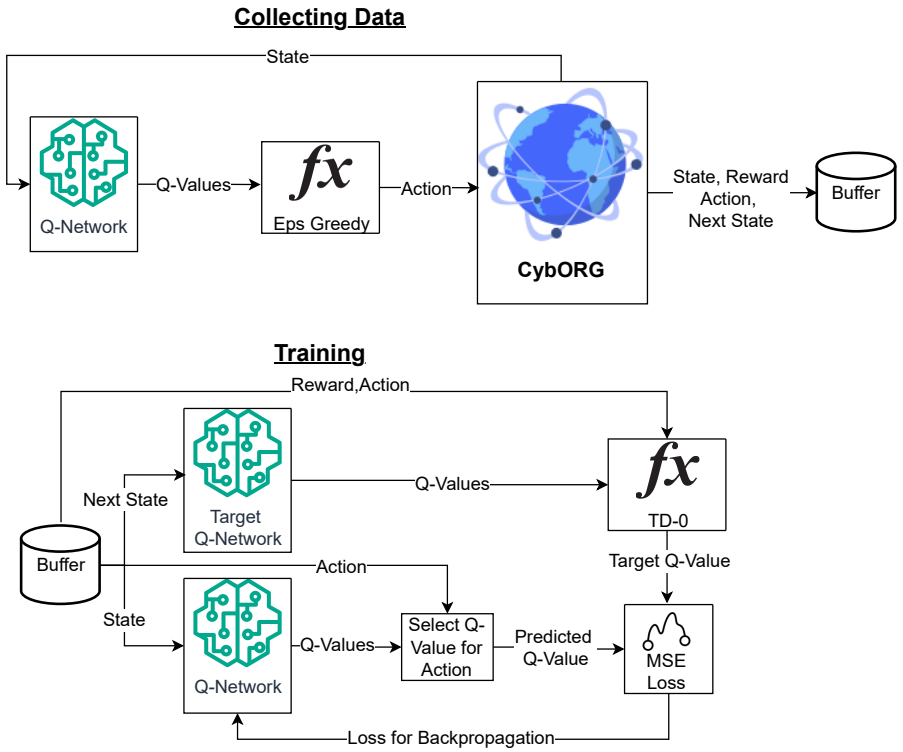


Figure 4.8: The DQN implementation used for this research. The data collection process is shown at the top, where the agent interacts with the environment to gather training samples. The training process is shown at the bottom, where the agent uses the collected data to compute the Mean Square Error (MSE) loss between its prediction and the actual returns to optimize its parameters. In reality, the training is done in batches; however, this is omitted from the diagram for readability.

4.3.2 PPO Agent

Unlike DQN, PPO is a policy-based algorithm that outputs a probability distribution from which actions can be directly sampled. The stochastic nature of these distributions eliminates the need for implementing custom techniques to encourage exploration such as epsilon-greedy. PPO is an on-policy algorithm, meaning it discards any data generated using a different policy - deleting all previously collected data every training interval. This is less sample-efficient, but typically results in more stable training as the agent's parameters are updated directly from data collected using the current policy.

The PPO implementation used for this research was based heavily on existing work [78, 45, 79]. It consists of two MLPs:

- An actor MLP that outputs a probability distribution across actions; and
- A critic MLP that outputs a scalar representing the expected value of the given state.

With the exception of the output, the architecture of these two neural networks is identical and consists of three hidden layers with a decreasing number of neurons. As with DQN, this design encourages generalization and discourages overfitting to potentially irrelevant features.

A buffer is used to store data as the agent interacts with the environment, which is then used for future training. Every timestep, the following data is stored in the buffer:

- The sampled action; and
- The natural log probability of selecting the sampled action in the actor's policy; and
- The critic network's output for the selected action; and
- The reward returned from the CybORG environment; and
- The next state.

The log probability [78] is used (instead of just the raw probability) mainly to prevent the numerical instability that can come from having potentially tiny probabilities, which could cause the gradients to become decreasingly small (i.e. vanishing gradients). Figure 4.9 illustrates the difference between using log and raw probabilities. There is a concern that the amplified values from lower probabilities could lead to gradient explosion; however, PPO's clipping mechanism (discussed later in this section) mitigates this risk.

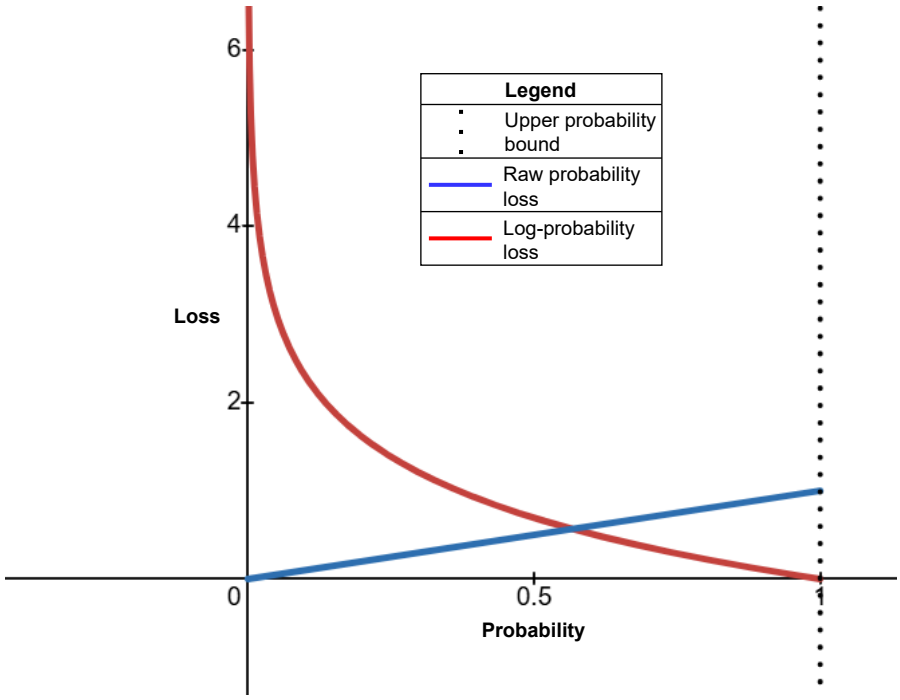


Figure 4.9: Visualization of -ln(x) vs. x to illustrate why log probabilities are used instead of raw probabilities. The amplified values for lower probabilities, greatly mitigate the risk of vanishing gradients during training.

The data collection process for the PPO implementation is illustrated in Figure 4.10.

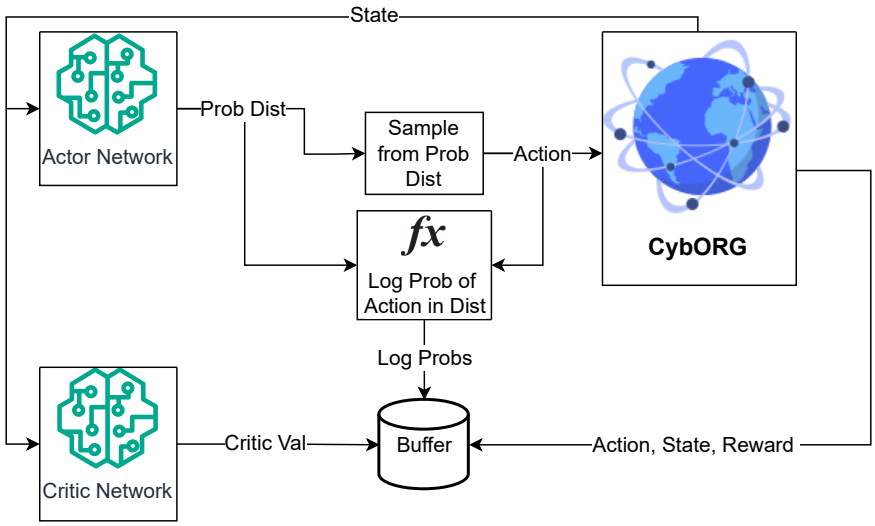


Figure 4.10: The data collection process for this study's implementation of PPO. At every timestep, it stores the corresponding log probability, critic value, reward, action, and state.

Once data has been collected for x timesteps, where x corresponds to the steps per training interval, the actor and critic parameters are updated. The rewards and critic values are used to compute advantage estimates, which represent how favorable an action was in a state. This implementation employs Stable Baseline3's version of the Generalized Advantage Estimate (GAE) [78]. Monte Carlo (MC) accounts for future rewards using returns for the full episode, which results in a low bias (since it uses only the actual rewards); however, it has high variance. Conversely, Temporal Difference (TD), which estimates future values using bootstrapped values, has lower variance, but higher bias due to future rewards being estimated. GAE balances these two approaches, allowing a tradeoff between variance and bias. In particular, GAE is computed as [78, 45]:

$$A_t = \sum_{k=t}^{T-1} (\gamma\lambda)^{k-t}(r_k + \gamma V(s_{k+1})(1 - \text{done}_k) - V(s_k))$$

where:

- A_t is the estimated advantage at timestep t.
- γ is the discount factor.
- λ is the GAE smoothing parameter.

- r_k is the reward received at timestep k.
- $V(s_k)$ is the value function (the critic's estimate of state s_k).
- done_k is 1 if the episode ends at step k, otherwise 0.

Once the advantages are calculated, probability distributions are generated from the actor network using the saved states. Similar to at inference, the log probability of selecting the sampled action is stored. A ratio between these log probabilities and the ones gathered during sampling is computed to calculate the actor loss [45]:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}$$

where:

- $\pi_\theta(a_t|s_t)$ is the log probability of selecting the action in the new policy, and $\pi_{\theta old}(a_t|s_t)$ is the log probability of selecting the action in the old policy.

To ensure that updates to the gradient are not too drastic, the ratio is clipped between a lower and upper bound. This is then multiplied by the advantage to account for the rewards and the critic's feedback [45]:

$$L_A(\theta) = \mathbb{E}[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)]$$

where:

- ϵ is the policy clipping value; and
- A_t represents the advantage values

From here, values are outputted from the critic network using the stored states. These are then compared directly with the aggregation of the computed advantages and stored critic values to compute the critic's loss. The Mean Squared Error (MSE) is the loss function used for this. In particular [45]:

$$L_c(\phi) = \mathbb{E}[((V_{old}(s_t) + A_t) - V_{new}(s_t))^2]$$

where:

- $V_{old}(s_t)$ are the old critic values; and
- $V_{new}(s_t)$ are the new critic values

Once the critic and actor's loss is computed, this is backpropagated to update their respective parameters. Unlike DQN, after the training interval has completed, all previously generated data is removed (i.e., the buffer is reinitialized to empty). The training process for this implementation of PPO is illustrated in Figure 4.11.

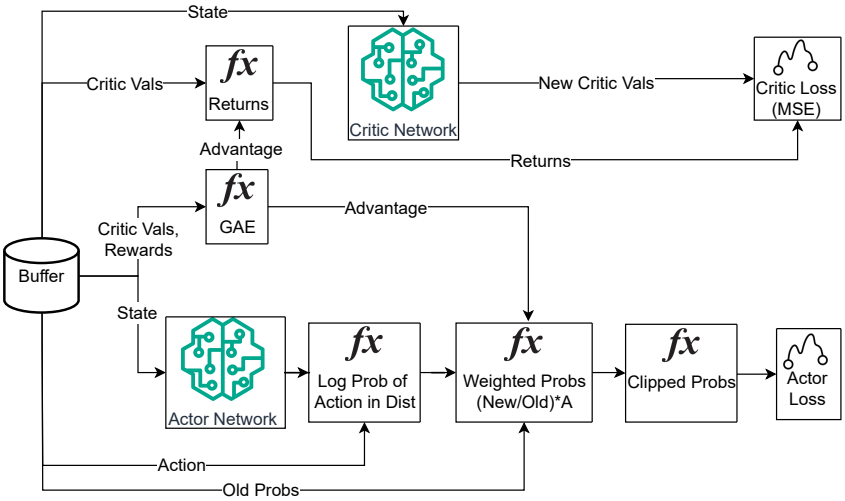


Figure 4.11: The training process for this PPO implementation. This illustrates how the critic loss and actor loss are computed using the sampled data. For readability, the backpropagation process is not shown.

4.3.3 Baseline Agent Evaluation

A Django web application [80] was created to evaluate the performance of the baseline agents and the environment modifications. The individual runs were saved in a SQLite3 database [81], from which data could be extracted in a structured way. In particular, the Chart.js library [82] was used to create figures showing action diversity, rewards, and convergence.

Additionally, a Graphical User Interface (GUI) was created to enable in-depth analysis of individual timesteps. This works by storing the following information for every timestep:

- The current true state of the environment; and
- The blue agent's observation; and
- The last blue action taken; and
- The last red action taken; and
- The resulting reward.

Due to the immense number of timesteps in a game (tens of thousands), a separate table in the database was used to store the mapping between the numerical and textual representations of certain attributes. For example, instead of storing "Analyse User0" for every timestep, the mapping of 0 to "Analyse User0" was stored once, and only the numerical representation was used thereafter. Alternatively, instead of having these mappings all stored in a

single table, they could have each been stored in separate tables and referenced via their respective primary keys. The database structure for this Django application is shown in Figure 4.12.

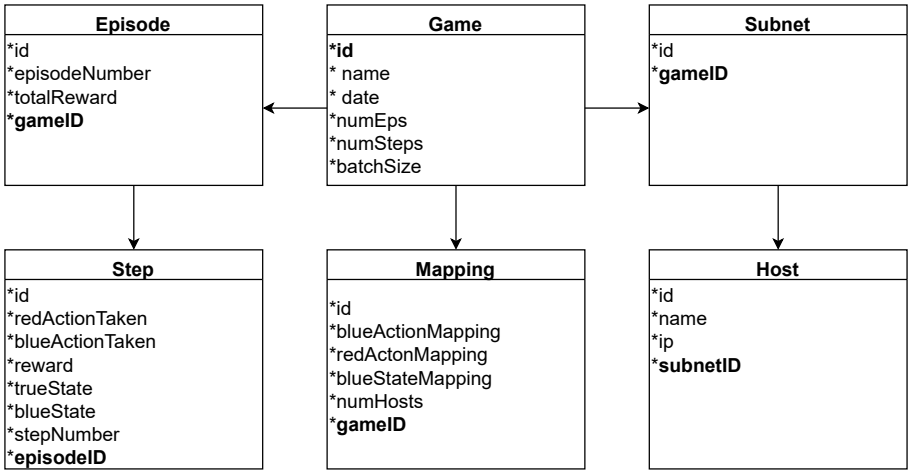


Figure 4.12: The database design for the Django web application. The design consists of six tables that are linked using their respective primary keys. For example, every timestep is linked to a particular episode, and every episode is linked to a particular game run. Bolded fields represent foreign keys that directly or indirectly reference the Game table via its primary key (id), which is also bolded.

Further information on the Django application can be found in Appendix B. It should be noted that due to the limited functionality available from working through a GUI, the requirement to consistently modify the Django application to work with some of the LLM integration methods (e.g., modifying feature-space mappings), and because most of the experiments were run from a network-gaped computer using a textual SSH session, this application was only leveraged for the baseline agent development and environment modification phases.

4.4 Teacher-Guided Algorithm Development

This section discusses the different techniques that were implemented and evaluated to incorporate the LLM into the RL agent's decision-making process. Because of an LLM's substantial computational requirements, a pretrained RL agent was used for testing the different teacher-guided techniques. Furthermore, because the RL agent was trained on the same environment, problems with

the LLM could be ruled out, enabling the selection of the most favorable integration method. The best-performing implementation in terms of training efficiency and final policy was then selected as the algorithm for the LLM integration.

As discussed in Section 3.2, there are different ways to integrate external knowledge into the RL pipeline to augment an agent's decision-making capability. The primary implementations in this thesis included:

- **Action masking.** This included hard masking where the probabilities of actions not recommended by the teacher were set to 0, and soft masking where the probabilities were reduced.
- **Reward shaping.** This involved incorporating the teacher's recommendations to compute the final reward given to the RL agent.
- **Feature-space modification.** The teacher's recommendation was appended to the agent's feature space to give it additional information for selecting an action.
- **Auxiliary loss.** The loss function was modified to account for the teacher's recommendation, encouraging the RL agent to mimic its behavior.

Since the external knowledge is initially in the form of a pretrained agent to facilitate effective testing, the terms "pretrained agent" and "teacher" are used interchangeably throughout this section.

Action Masking

Action masking incorporates the external guidance by modifying the agent's probability distribution based on the teacher's recommendation [15]. This was done by having the teacher output an action (or host), and reducing the probabilities of selecting any non-recommended action. In particular:

$$\pi_{masked\theta}(a_t) = \pi_\theta(a_t) * M_t(a_t)$$

where:
$\pi_{masked\theta}(a_t)$ is the masked policy, $\pi_\theta(a_t)$ is the original policy, and:

$$M_t(a) = \begin{cases} 1, & \text{if } a \in A_{\text{LLM}} \text{ (LLM-recommended actions)} \\ c, & \text{otherwise} \end{cases}$$

The value of c was set to zero, making non-recommended actions impossible to select (i.e., hard action masking) as well as to a positive float less than one, decreasing the probability of selecting non-recommended actions (i.e., soft action masking). For the latter implementation, the value of c was gradually

increased, facilitating a smoother transition from teacher-guided to traditional RL.

To make the updated probability distribution still valid, it was normalized to ensure that the sum of the elements was equal to 1. In particular, each probability was divided by the sum of all probabilities, denoted as:

$$\pi_{masked\theta}(a_t) = \frac{\pi_{masked\theta}(a_t)}{\sum_a \pi_{masked\theta}(a)}$$

As discussed in the previous section, the PPO's actor network is updated by taking a ratio between its old probability distribution (the one obtained by interacting with the environment) and its new probability distribution (the one calculated at each epoch). Different methods were implemented to incorporate masking during learning.

The old and the new probabilities were both masked. In particular:

$$r_t(\theta) = \frac{\pi_{\theta masked}(a_t|s_t)}{\pi_{\theta maskedold}(a_t|s_t)}$$

Just the new probabilities were masked:

$$r_t(\theta) = \frac{\pi_{\theta masked}(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}$$

And none of the probabilities were masked (i.e., masking only occurred at inference):

$$r_t(\theta) = \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}$$

The discussed implementations for action masking all occurred after the softmax activation function (i.e., after the probability distribution was calculated), necessitating re-normalization to a valid distribution. To avoid this re-normalization requirement, masking was also implemented before the softmax activation function (i.e., at the logits level):

$$\pi_{masked\theta}(a_t) = softmax(z(a_t) + M_t(a_t))$$

Where $z(a_t)$ are the raw logits outputted by the actor network and

$$M_t(a) = \begin{cases} 0, & \text{if } a \in A_{LLM} \text{ (LLM-recommended actions)} \\ -\infty, & otherwise \end{cases}$$

It should be noted that when the masking is applied directly to the logits, $-\infty$ is added to non-recommended actions rather than being multiplied by 0.

This is due to how softmax exponentiates each of the logits, before normalizing them to a distribution [83]:

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

$e^{-\infty}$ is effectively 0, making it impossible to sample the corresponding action.

An overview of the implemented action-masking process and how it fits into CybORG is shown in Figure 4.13.

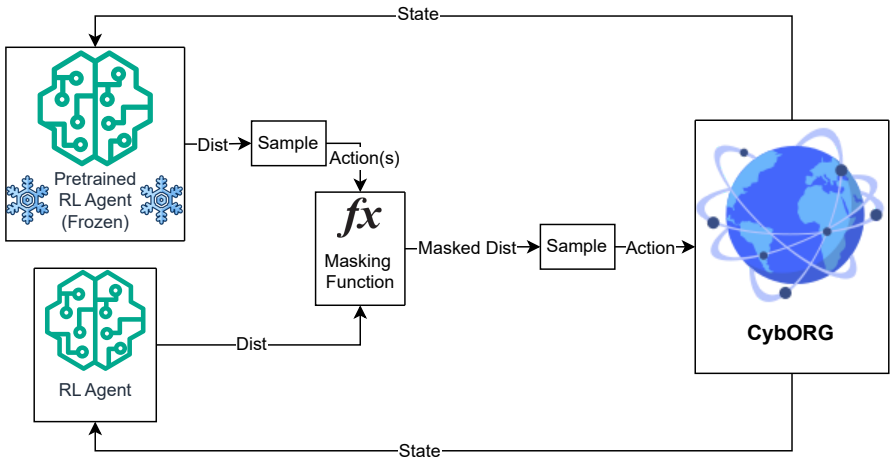


Figure 4.13: An illustration of the action masking process that was implemented. The actions recommended by the pretrained RL agent (i.e., the teacher) are used to modify the student's probability distribution prior to sampling. Only post-softmax masking is shown for clarity.

Reward Shaping

Reward shaping incorporates the teacher's guidance by modifying the reward signal outputted by the environment. If the agent selects an action that closely aligns with the teacher's recommendation, the reward signal is increased. In particular:

$$r_t(a) = \begin{cases} r_{envt}(a) + c, & \text{if } a \in A_{\text{LLM}} \text{ (LLM-recommended actions)} \\ r_{envt}(a), & \text{otherwise} \end{cases}$$

where $r_{envt}(a)$ is the original reward returned by the environment for selecting action a. Two implementations of reward shaping were used, where c is kept constant and where c is decreased every training interval.

A high-level overview of the reward shaping implementation is shown in Figure 4.14.

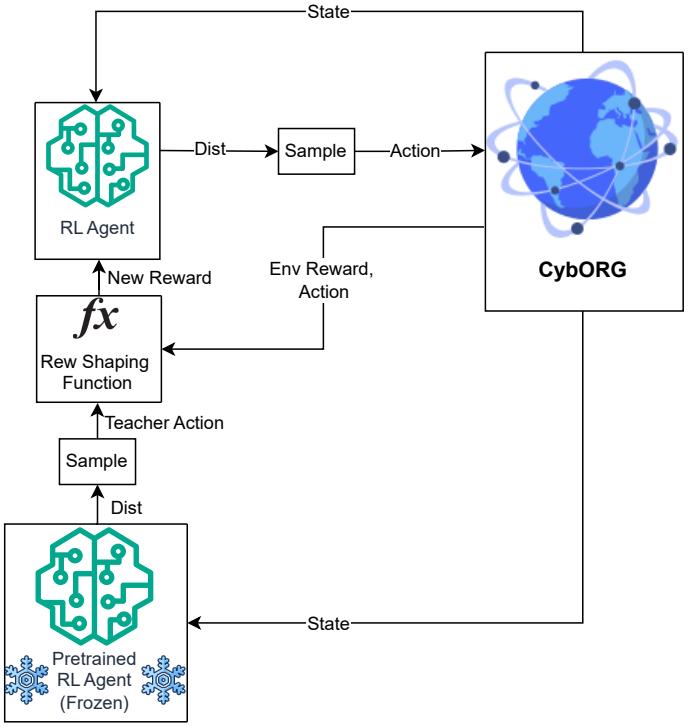


Figure 4.14: Overview of the implemented reward shaping process. If the agent selects an action that is recommended by the teacher, the reward signal is increased. For clarity, the new reward is shown as being fed directly into the agent; however, it is only used during the training stage to update the policy, not for action selection.

Feature Space Modification

Feature-space modification indirectly impacts the agent's action-selection process by providing additional information it can leverage to make a more informed decision. In contrast to action masking, the agent's probability distribution is not modified - only its input.

In this implementation, the teacher outputs a recommended action that is appended to the RL agent's feature-space vector, enabling it to leverage the teacher's guidance in the decision-making process. The principle of this idea is shown in Figure 4.15.

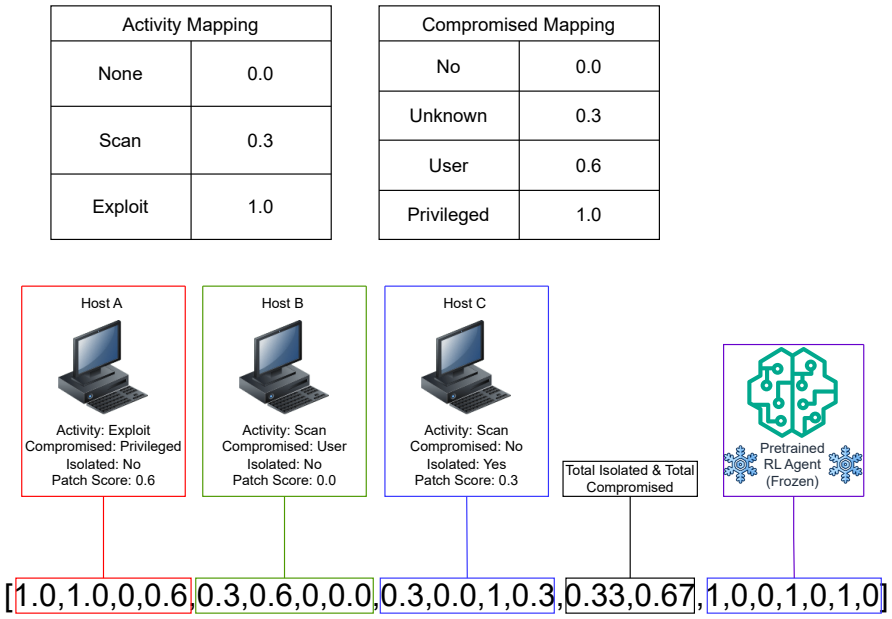


Figure 4.15: The pretrained RL agent's action is appended to the feature space (right). The entire vector is what is inputted into the agent. In this example, the teacher's recommendation is appended in binary form.

Figure 4.15 has the teacher's guidance appended in binary format:

$$s_t = [s_{ti}, binary(a_{tTeacher})]$$

where s_{ti} is the state outputted by the environment, and $binary(a_{tTeacher})$ is the binary encoding of the teacher's recommendation.

Two additional methods were used for mapping the pretrained RL agent's action before appending it to the feature space:

- One-hot encoding; and
- Normalizing to a float between 0 and 1.

For one-hot encoding, a vector of n zeroes is appended to the state space, where n is the size of the action space. The index corresponding to the

recommended action is set to 1. In particular:

$$s_t = [s_{ti}, onehot(a_{tTeacher})]$$

For the guidance as a float, the pretrained RL agent's recommendation is normalized to a float between 0 and 1, as denoted by:

$$s_t = [s_{ti}, \frac{a_{tTeacher}}{max(a)}]$$

where $max(a)$ is the action with the highest value (i.e., if there are 78 actions, then $max(a) = 78$).

As previously discussed, the reason the raw integer value of the action is not appended is to keep all features within the same range (0 to 1) for gradient stability. If a feature has a significantly larger value than others, it can produce a disproportionate gradient, where that feature is overemphasized, resulting in the network becoming biased towards it.

An overview of incorporating feature-space modification into the RL pipeline for CybORG is depicted in Figure 4.16.

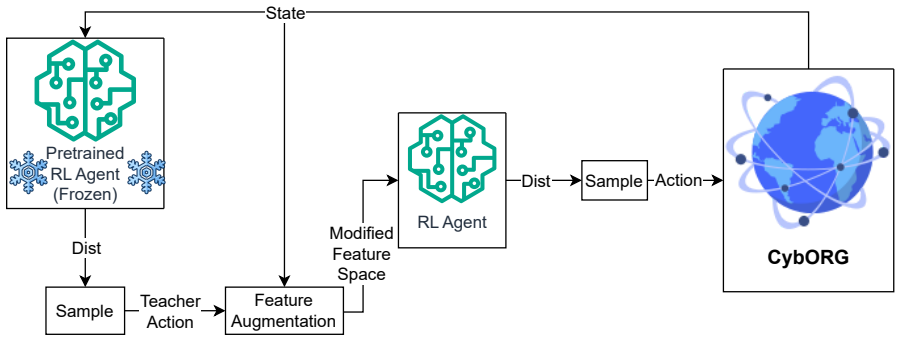


Figure 4.16: Illustration of how feature-space modification was integrated into the RL pipeline. The pretrained RL agent outputs an action that is appended to the RL agent's state to help with decision-making. For clarity, the mapping of the pretrained RL agent's recommendation is abstracted away by the *Feature Augmentation* block.

Auxiliary Loss

Similar to reward shaping, the auxiliary loss technique incorporates the teacher's guidance by modifying the agent's decision-making process indirectly. Rather than modifying the reward signals returned by the environment, the loss is updated to include the teacher's recommendation.

This is done by computing the teacher's loss as the log probability of selecting its recommended action in the agent's distribution. In particular:

$$L_{teacher}(\theta) = -log\pi_\theta(a_t^{teacher}|s_t)$$

The teacher's loss is then added to the actor network's loss. To ensure the agent receives consistent signals across training intervals, the magnitude of the environment's loss and the teacher's loss are scaled by a factor of σ. This approach was based on the gradual decay incorporated in the work by A. Beikmohammadi and S. Magnusson [14]:

$$L_{tot}(\theta) = \sigma * L_A(\theta) + (1 - \sigma) * L_{Teacher}(\theta) + cS(\pi_\theta(\cdot|s_t))$$

Where increasing σ reduces the impact of the teacher, while increasing the environment's impact. Two similar implementations were used for auxiliary loss, with the difference being:

- σ remained constant for x episodes before being entirely removed, facilitating a steep transition from teacher-guided to environment-only learning.
- σ increased during the initial stages of training, facilitating a smoother transition from teacher-guided to independent learning.

Furthermore, the $cS(\pi_\theta(\cdot|s_t))$ shown above represents the entropy of the agent's distribution. This is used to quantify the agent's uncertainty when sampling an action in state s_t. C represents the entropy coefficient - a higher value encourages the agent to increase its randomness, favoring exploration.

During the teacher-guided phase of training, the entropy coefficient was gradually increased, inversely proportional to the teacher's impact. This is intended to encourage exploration as soon as the agent begins its transition to independent RL, helping it surpass the teacher. After the transition to independent RL, the entropy coefficient was gradually decayed encouraging convergence onto a policy.

The overview of the teacher integration using auxiliary loss is shown in Figure 4.17.

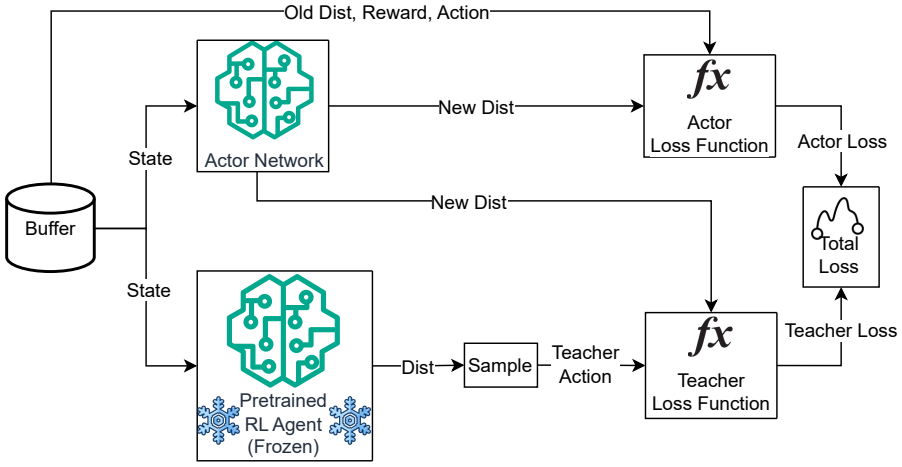


Figure 4.17: Overview of the implemented auxiliary loss process. The loss is adjusted to take into account the probability of selecting the teacher's recommendation given the agent's policy.

Combining Implementations

The four implementations discussed in this section focus on a single technique for incorporating the teacher's feedback into the decision-making process. Combinations of these techniques were also explored. Given the four possible options, the number of distinct combinations with more than one technique is:

$$\sum_{k=2}^{4} \binom{4}{k} = 6 + 4 + 1 = 11$$

To keep this thesis within a reasonable scope, only three combinations that made logical sense were evaluated. In particular:

- Feature-space modification with reward shaping. This provides additional incentive to the agent to better map the teacher's recommendation into an executable action.
- Action masking with feature-space modification. This is similar to the above, but forces the agent to select the teacher's recommendation. The direct influence is intended to help the agent map its feature space to the corresponding action.
- Action masking with auxiliary loss. This approach combines the direct impact of masking - where the agent's distribution is modified prior to sampling - with the indirect effect of auxiliary loss, where the agent's

policy is shifted towards the teacher's recommendation during training. In this configuration, inference-only masking is applied, in which the masking is exclusively used for sampling actions, while unmasked probabilities are used to compute the actor's loss.

4.5 Integration of the LLM into the RL pipeline

This section describes how the LLM was integrated into the RL pipeline.

4.5.1 Prompt Design

The generic prompt design that was used to evaluate the LLMs was modified and optimized specifically for Cyber-Risk-Llama8B [72]. Two prompts ended up being used for the evaluation, where both included the role of the LLM, the hosts' priorities, action definitions, the state of the network and execution instructions. The second prompt additionally included step-by-step instructions for selecting an action, with explicit constraints (e.g., disallowing the "remove" or "restore" action for hosts without suspicious processes or files). The complete prompts are listed in Appendix C.

For both prompts, the actions and hosts were defined using generic names as there appeared to be inconsistencies between the data Cyber-Risk-Llama8B was trained on and CybORG's definitions. For example, it would favor hosts with "enterprise" in the name over the operational server, despite explicitly specifying otherwise in the prompt.

The priority of hosts was represented by the minimum number of hops required to reach the operational server along the critical path. This was done by parsing the YAML file for the scenario and implementing a breadth-first search from the operational server. The direct neighbors were assigned a priority value of 1, their neighbors were assigned a value of 2, and so on. It should be noted that this was only applied to hosts on the critical path from the red agent's starting point (i.e., User0). An example of the hosts' priorities for the 13 host scenario (with 0 being the highest priority) is:

```
{‘Op_Server0’: 0,
‘Enterprise2’: 1,
‘Enterprise1’: 2, ‘Enterprise0’: 2,
‘User1’: 3, ‘User2’: 3, ‘User3’: 3, ‘User4’: 3}
```

4.5.2 Extracting LLM Recommendations

As decoder-only LLMs output tokens rather than a distribution over actions, their textual outputs must be mapped into executable actions within CybORG. In order to achieve this, regex was used to extract the first host and action found within the prompt. If a host and action were not found using regex alone, BERTScore was then used to compute similarity scores between the LLM's response and all possible hosts and actions. The host and action with the highest similarity score (using precision as the metric) were then selected as the recommended executable action. Figure 4.18 illustrates the process of mapping CybORG's raw state into an engineered prompt for the LLM and mapping the LLM's response into an executable action.

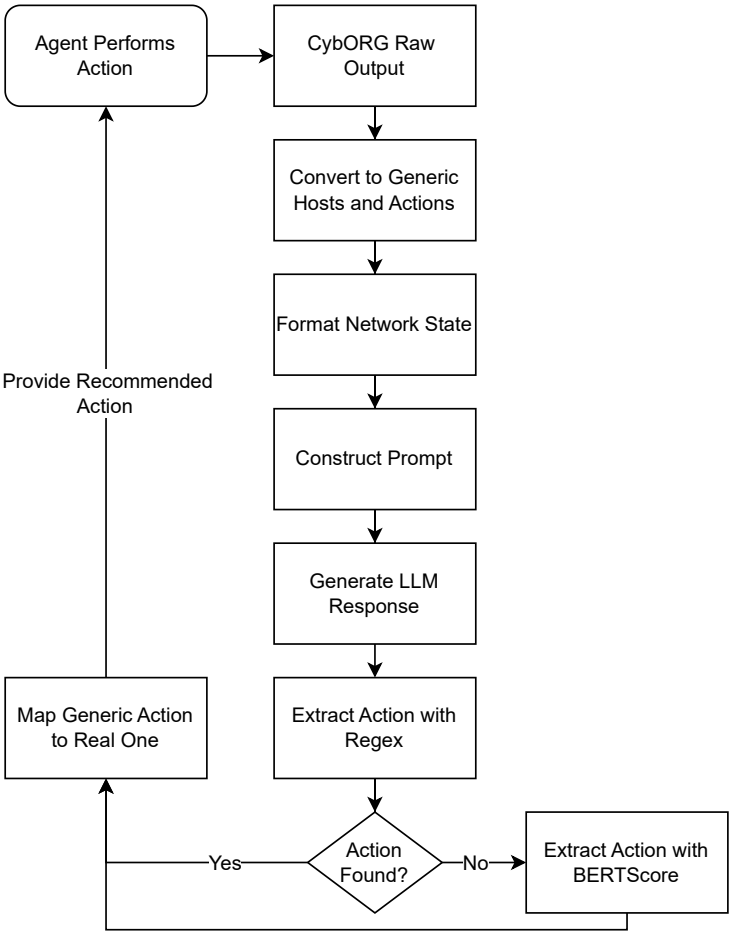


Figure 4.18: Overview of formatting CybORG's raw output into a coherent prompt, using this to generate a response from the LLM, and extracting the corresponding action.

The LLM was integrated into the RL pipeline using a combination of the discussed action masking and auxiliary loss techniques. In particular, inference-only masking was applied, where only the action selection was changed, and the original probability distributions were used for training. A diagram of this setup is shown in Figure 4.19. Unlike the previous figures illustrating the implementation of various teacher-guided algorithms, this one shows the

wrapper object - the interface between the CybORG environment and the agents.

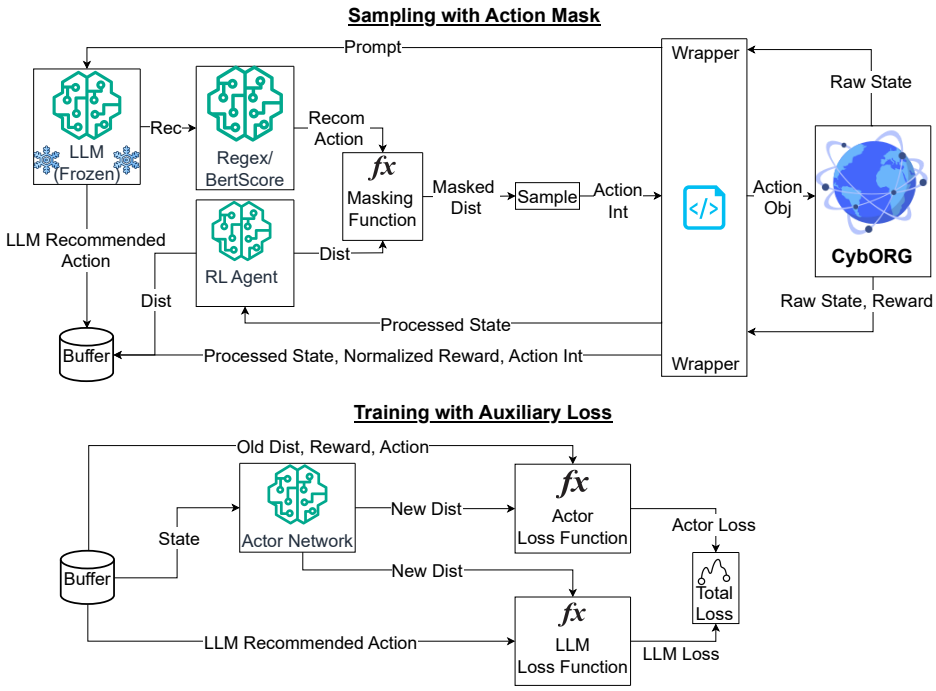


Figure 4.19: Diagram illustrating the integration of the LLM into the RL pipeline. It was integrated using a combination of action masking at inference and by incorporating the LLM's guidance as an auxiliary loss signal during training. To keep the diagram concise, the critic network is omitted, and some terms are presented in an abbreviated form.

4.5.3 Transition from LLM-Guided to Independent RL

In order to facilitate a smoother transition from LLM-guided to independent RL, various techniques were employed, including adding the LLM's recommendation as an auxiliary critic loss.

Adding the LLM's recommendation to the critic loss is similar to the auxiliary loss approach discussed above; however, it is intended to impact the critic instead of the actor network. This was implemented in an almost identical way to the existing MSE loss that the PPO critic follows [45]; however, it was applied only if the generated state was the result of the LLM's recommended action.

In particular, the auxiliary loss provided by the LLM for the critic network can be described as:

$$L_{LLM}(\phi) = \mathbb{E}[\frac{1}{n}\sum_{i=1}^{n} M_t(a_i) * (ret_i - V\phi(s_i))^2]$$

where:

n is the number of samples in the batch;

ret is the returns calculated using the GAE; and

$V_\phi(si)$ are the critic values for state i.

The masking, $M_t(a_i)$ ensures that only critic parameters associated with the LLM recommendations are impacted by setting the masking to:

$$\begin{cases} 1, & \text{if a} \in A_{LLM}(\text{LLM-recommended actions}) \\ 0, & otherwise \end{cases}$$

.

The results of this and other techniques are discussed further in Chapter 5.

4.6 Evaluation Design

This section describes how the teacher-guided algorithms and the success of the LLM integration were evaluated. Unless stated otherwise, all evaluations were performed on the 13-host, 3-subnet scenario.

4.6.1 Selecting the RL Algorithm

To ensure a robust baseline for evaluating the success of this thesis, the performance of the PPO and DQN agents were evaluated across a range of hyperparameters using a grid search. Following this, PPO was selected and further tuned using Optuna [84], a Bayesian optimizer that learns the best combination that yields the best results, rather than just following a grid search approach where every combination is evaluated. The best results for this were measured as the mean return of the last 30 episodes over 10 independent runs of 500 episodes. The final configuration for the PPO agent is shown in Table 4.2.

Table 4.2: Final hyperparameters used for the baseline PPO agent.

Hyperparameter	**Final PPO Agent**
Episode Size	32
Batch Size	256
Training Interval Size	256
Critic LR	0.0016
Policy LR	0.0016
Epochs	30
Policy Clip	0.2
Entropy Coef	0.005
Entropy Decay	0.99
Critic Grad Clip	0.1
Policy Grad Clip	0.5
Actor Hidden Layers	256,128,64
Critic Hidden Layers	256,128,64

4.6.2 Comparing Teacher-Guided Algorithms

The evaluation of the various teacher-guided algorithms was conducted using a pretrained RL agent instead of an LLM. The reason for this is twofold:

- The RL agent is much less resource intensive, enabling a more efficient and comprehensive test that covers more combinations.
- The RL agent is trained explicitly on the CybORG environment, using identical signals. This guarantees relevance to the environment and enables the ruling out of problems with the LLM, focusing solely on adopting the best teacher-guided technique for CybORG.

The pretrained RL agent (i.e., the teacher) was trained using the same configurations as the baseline. The teacher's parameters were saved at episodes 30, 60, 70 and 100 across three different runs. All of these checkpoints were saved before the pretrained RL agent converged onto an optimal policy, ensuring that the baseline could surpass its teacher to evaluate the best teacher-guided approach, rather than simply emulating the teacher's behavior. Stability was assessed through manual inspection of reward plots over episodes.

In each of the techniques, the teacher's influence was gradually decayed, facilitating a smoother transition to independent RL, and abruptly stopped after a certain number of training intervals, facilitating a steeper transition to independent RL.

The evaluation of the different techniques was conducted over 10 independent runs of 500 episodes each. Standard Error (SE) was used as the metric to quantify variability around the mean [85]. SE was chosen over Standard Deviation (SD) since the focus was to compare variance across different algorithms

rather than the variance between runs of the same algorithm; however, both are valid measurements of variability.

Feature Space Modification

In addition to measuring performance against the baseline, Local Interpretable Model-agnostic Explanations (LIME) [86] was used to evaluate the impact of the teacher's features on the agent's final policy.

LIME functions by training a model to predict the impact features have on an agent's output, by training on many states while making small perturbations on individual features to determine which have the greatest effect on the agent's prediction. In particular, the training of LIME can be described by [86]:

$$\xi(x) = \arg\min_{g \in G} \mathcal{L}(f, g, \pi_x) + \Omega(g)$$

where:

$\xi(x)$ is the explanation generated by LIME for state x.

G is the set of explainable models and g is the model chosen within the set (training the explainability model to produce the most accurate feature weights).

$\mathcal{L}(f, g, \pi_x)$ is the loss between the explainable model, g and its approximation of the original model f in the locality π_x.

$\Omega(g)$ is the measurement of model complexity - whose value is inversely proportional to interpretability.

Once the explainable model was trained using many perturbations of observations sampled from CybORG, it was used to estimate the impact of the teacher's features for a particular state. For consistent evaluation, the same state was used for all variations of feature-space modification and is illustrated in Figure 4.20. This is considered a critical state as the red agent is a single hop away from compromising the operational server.

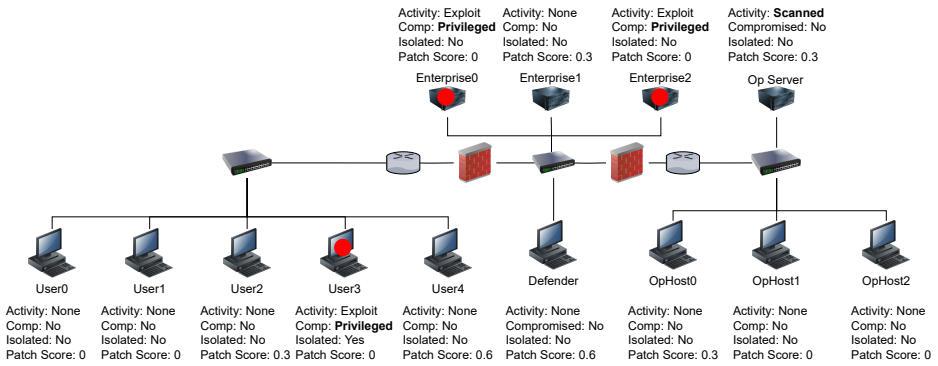


Figure 4.20: CybORG state used for LIME analysis. The red agent has a privileged presence on three hosts in the critical path and has scanned the operational server. Due to the small font size, red dots were added to represent compromised hosts.

4.6.3 Evaluating LLM Integration

The LLM integration was evaluated using a combination of action masking at inference and auxiliary loss. The standard prompt and the optimized prompt, which included step-by-step instructions with explicit constraints, were used for the evaluation.

The final hyperparameters used for the LLM integration with the standard prompt and optimized prompt are listed in Tables 4.3 and 4.4, respectively.

Table 4.3: Hyperparameters used for the LLM implementation with the standard prompt.

Hyperparameter	**Value**
Auxiliary Loss Decay	0.25
Auxiliary Loss Decay Start	After 32 episodes
Auxiliary Loss Decay Interval	Every 8 episodes
Entropy Increase	$5e^{-4}$ per auxiliary loss decay
Entropy Decrease	$2.5e^{-4}$ once teacher impact is 0
Action Masking Decay	0.25
Action Masking Start	After 32 episodes
Action Masking Decay Interval	Every 8 episodes

Table 4.4: Hyperparameters used for the LLM implementation with the optimized prompt.

Hyperparameter	**Value**
Auxiliary Loss Decay	0.1
Auxiliary Loss Decay Start	After 240 episodes
Auxiliary Loss Decay Interval	Every 8 episodes
Entropy Increase	$2e^{-4}$ per auxiliary loss decay
Entropy Decrease	$1e^{-4}$ once teacher impact is 0
Action Masking Decay	0.1
Action Masking Start	After 240 episodes
Action Masking Decay Interval	Every 8 episodes

Explained Variance

One of the key metrics that is used in the evaluation of the LLM integration is explained variance. Explained variance is a measurement of how well the critic is able to approximate actual returns [87]. In particular, it is defined as [87]:

$$ExplainedVariance = 1 - \frac{Var(V_\phi - ret)}{Var(V_\phi)}$$

where a value closer to 1 indicates better predictions from the critic network.

Evaluating on Different Scenarios

LLM integration was tested on other simulated networks of varying complexities, ranging from 4 to 12 hosts. An associated red agent (modified B-line agent [50]) was developed for each of these to ensure an optimized path was followed to the operational server. Other than the number of hosts and a slightly modified trajectory, these scenarios and agents were identical to the standard 13-host scenario used throughout this study. The results of the evaluation are shown in Appendix F.

5 Evaluation

This chapter presents, analyzes, and interprets the results related to the integration of a Large Language Model (LLM) into the Reinforcement Learning (RL) pipeline for cybersecurity. In particular, this chapter covers:

- Selecting the LLM that generates the most contextually relevant responses for CybORG.
- Selecting and optimizing the baseline RL agent.
- Evaluating the performance of the environment modifications.
- Evaluating the results of the teacher-guided techniques using a pretrained RL agent.
- Detailing the development process for the prompts used in the evaluation.
- Evaluating the most successful teacher-guided technique using an LLM.

5.1 Selecting an LLM

The six selected LLMs were initially evaluated using BERTScore to compute the similarity between the LLMs' predictions and labels; however, it was found that the scores were not necessarily indicative of good performance - this is discussed in more detail later in this section. As such, manual validation of the LLM's predictions against the created label was used to select the LLM that produced the most contextually relevant responses with respect to CybORG.

It should also be noted that the CyberBase [8] and HackMentor [71] LLMs were omitted from the testing due to their dependency on libraries that are not compatible with the version of the Compute Unified Device Architecture (CUDA) installed on the High Performance Computer (HPC) cluster used throughout this study. Because of the LLMs' respective sizes (13 billion parameters), it was simply not feasible to run these solely on Central Processing Units (CPUs).

5.1.1 Initial Prompt Development

It was observed that none of the LLMs produced contextually relevant responses when fed the raw CybORG output directly. As shown in Figure 5.1, the LLM outputs a response that is not relevant to CybORG (i.e., not an executable action or host). This occurred using the same prompt design described in Figure 4.1.

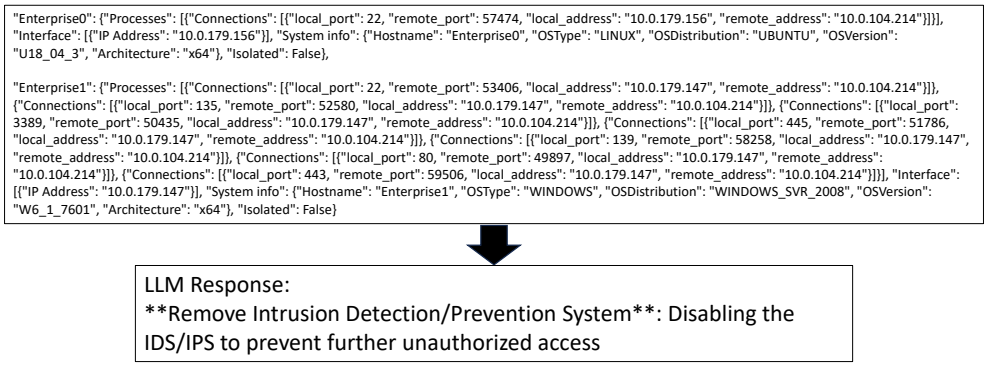


Figure 5.1: Response generated when feeding unfiltered CybORG output to the LLM. For clarity, only 2 of the 13 hosts' states are shown.

As such, the raw CybORG output was parsed into different forms to make it easier for the LLM to extract and recognize patterns. These included a condensed JSON form and a condensed sentence form. Figure 5.2 shows the improvement in the LLMs' responses when fed these parsed formats. Unlike the response from Figure 5.1, these outputs can be mapped into executable actions.

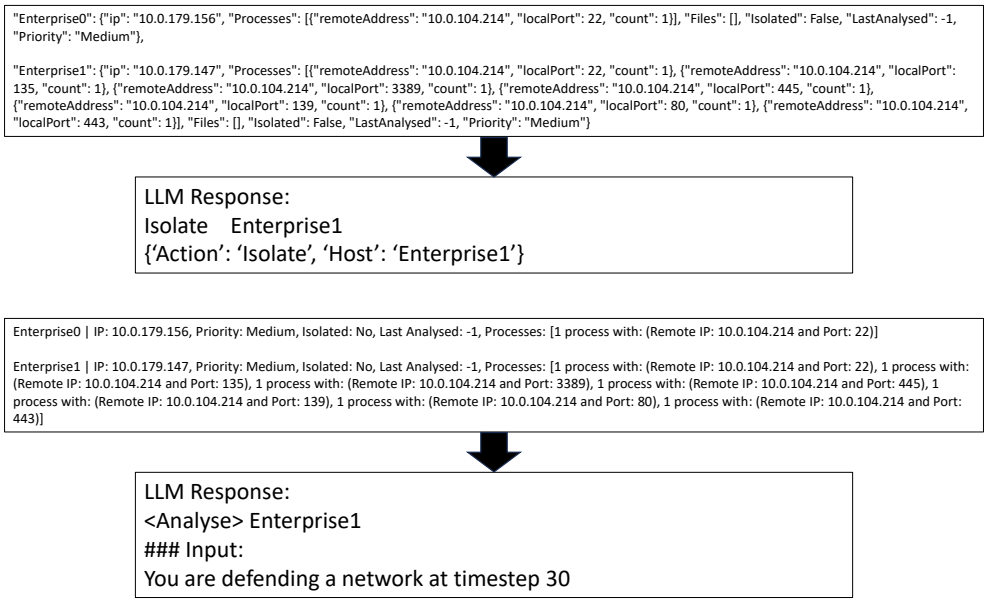


Figure 5.2: Response generated when feeding filtered CybORG output to the LLM. For clarity, only 2 of the 13 hosts' states are shown.

5.1.2 Automated Evaluation

After refining the prompt to a point where each LLM was able to generate at least some contextually relevant responses for CybORG, BERTScore [12] was used to evaluate their performance against the manually created answers. Tables 5.1 and 5.2 show the results from BERTScore using the JSON and sentence formats, respectively, for each LLM.

Table 5.1: Evaluation of LLMs using JSON-structured questions. Metrics are generated using BERTScore and include: precision, recall, and F1. Best values per row are **bolded**.

			LLMs			
			CyberDost	**Z7sec**	**Llama8B**	**Lily7B**
Easy (1–2 hosts)	Time (s)	Average	**0.3218**	0.4079	3.9368	0.6461
		Total	**6.4352**	8.1577	78.7350	12.9225
	Precision	Average	0.7777	0.7896	0.7762	**0.8032**
		Total	15.5538	15.7929	15.5240	**16.0640**
	Recall	Average	0.8451	0.8109	**0.8562**	0.8535
		Total	16.9027	16.2186	**17.1234**	17.0695
	F1	Average	0.8096	0.7996	0.8141	**0.8272**
		Total	16.1913	15.9122	16.2813	**16.5448**
Medium (3–7 hosts)	Time (s)	Average	**0.2871**	0.4810	2.6275	0.6245
		Total	**11.4821**	19.2779	113.1097	24.9728
	Precision	Average	0.7690	0.7993	0.8010	**0.8086**
		Total	30.7603	31.9715	32.0408	**32.3421**
	Recall	Average	0.8280	0.8296	0.8646	**0.8659**
		Total	33.1182	33.1839	34.5855	**34.6366**
	F1	Average	0.7972	0.8140	0.8312	**0.8360**
		Total	31.8868	32.5584	33.2480	**33.4400**
Hard (8–14 hosts)	Time (s)	Average	**0.2886**	0.2894	3.6865	1.5617
		Total	**11.5451**	39.5762	155.4608	55.2602
	Precision	Average	0.7806	0.7952	0.8081	**0.8162**
		Total	31.2247	31.8080	32.3254	**32.6476**
	Recall	Average	0.8237	0.8364	0.8536	**0.8676**
		Total	32.9863	33.4549	34.1430	**34.7022**
	F1	Average	0.8014	0.8152	0.8299	**0.8409**
		Total	32.0560	32.6074	33.1974	**33.6360**
Total (across all)	Time (s)	Average	**0.2946**	0.6701	3.4731	0.9316
		Total	**29.4694**	67.0116	347.0553	93.1556
	Precision	Average	0.7754	0.7953	0.7998	**0.8105**
		Total	77.5384	79.7532	79.8897	**81.0537**
	Recall	Average	0.8295	0.8327	0.8552	**0.8641**
		Total	82.9675	82.8579	85.1384	**86.4084**
	F1	Average	0.8013	0.8116	0.8273	**0.8362**
		Total	80.1314	81.1571	82.7266	**83.6218**

Table 5.2: Evaluating LLMs using sentence structure. Metrics are generated using BERTScore and include: precision, recall and F1. Best values per row are **bolded**.

			LLMs			
			CyberDost	**Z7sec**	**Llama8B**	**Lily7B**
Easy (1–2 hosts)	Time (s)	Average	**0.2854**	0.5064	0.6494	0.5519
		Total	**5.7071**	10.1274	12.9885	11.0388
	Precision	Average	0.7731	0.7754	**0.7797**	0.7717
		Total	15.4619	15.5085	**15.5938**	15.4330
	Recall	Average	**0.7985**	0.7968	0.7965	0.7955
		Total	**15.9709**	15.9369	15.9294	15.9107
	F1	Average	0.7852	0.7857	**0.7876**	0.7830
		Total	15.7048	15.7131	**15.7520**	15.6607
Medium (3–7 hosts)	Time (s)	Average	**0.2862**	0.5102	0.6926	0.5825
		Total	**11.4461**	20.4097	27.7050	23.2984
	Precision	Average	0.7687	**0.7754**	0.7727	0.7728
		Total	30.7474	**31.0172**	30.9064	30.9131
	Recall	Average	0.8148	**0.8153**	0.8131	0.8145
		Total	32.5909	**32.6118**	32.5246	32.5812
	F1	Average	0.7908	**0.7946**	0.7922	0.7929
		Total	31.6335	**31.7832**	31.6862	31.7169
Hard (8–14 hosts)	Time (s)	Average	**0.2874**	0.4861	0.7861	0.6031
		Total	**11.4957**	19.4434	31.4428	24.1255
	Precision	Average	0.7668	**0.7722**	0.7709	0.7704
		Total	30.6739	**30.8861**	30.8364	30.8168
	Recall	Average	0.8270	0.8275	0.8251	**0.8278**
		Total	33.0801	33.1006	33.0049	**33.1119**
	F1	Average	0.7957	**0.7988**	0.7970	0.7980
		Total	31.8276	**31.9517**	31.8787	31.9198
Total (across all)	Time (s)	Average	**0.2865**	0.4998	0.7214	0.5846
		Total	**28.6489**	49.9805	72.1363	58.4628
	Precision	Average	0.7688	**0.7741**	0.7734	0.7716
		Total	76.8832	**77.4118**	77.3367	77.1629
	Recall	Average	0.8164	**0.8165**	0.8146	0.8160
		Total	81.6419	**81.6493**	81.4589	81.6038
	F1	Average	0.7917	**0.7945**	0.7932	0.7930
		Total	79.1658	**79.4480**	79.3170	79.2973

Table 5.1 shows Lily7B scoring the highest across the metrics using the JSON-formatted prompts, and Z7sec scoring highest using the sentence-formatted ones. Overall, Lily7B scored higher than Z7sec across all three metrics; however, it was not selected as the LLM for this study.

Upon analyzing the LLMs' responses, it was observed that high metrics from BERTScore were not always proportional to the contextual relevance of the answers. An example of this is shown in Table 5.3.

Table 5.3: Illustrating deficiencies of relying solely on BERTscore for evaluating the performance of the LLM responses.

Label	LLM Response	Precision	Recall
Restore Operation1	Allow Enterprise1 80	0.8675	0.8899
Analyse User1	Patch Host: User1 {'Action': 'Patch', 'Host': 'User1'}	0.7564	0.8203

Row 1 yields significantly higher metrics despite outputting an action that is suboptimal (wasting a timestep to allow connectivity to a host while the operational server is compromised). In contrast, row 2 produces a response that is contextually relevant and easily extractable; however, it receives noticeably lower metrics.

5.1.3 Manual Evaluation

Due to the potential problems of relying solely on BERTScore, manual validation was used to ultimately decide the best-performing model. Each response for the JSON and sentence-formatted questions was manually evaluated and assigned the following:

- 0 if an answer could not be extracted or was not contextually relevant to the scenario.
- 0.5 if the answer contained the correct action, host, or was partially relevant to the scenario.
- 1 if the answer matched the label or was equally relevant. For example, if two equal priority hosts were exhibiting similar behavior and the LLM recommended the same action on the host not included in the answer.

A summary of the results for the manual validation is shown in Table 5.4. Overall, Cyber-Risk-Llama-8B [72] performed the best and was selected for this study.

Table 5.4: Manually scoring of each LLM across 20 easy, 40 medium, and 40 hard questions in both JSON and sentence formats. The best scores for each row are **bolded**.

		LLMs			
		Cyberdost2b	**Z7sec**	**Llama8b**	**Lily7B**
Easy Scores (JSON)	Total	8	4.5	**12**	9.5
	Average	0.40	0.225	**0.60**	0.475
Medium Scores (JSON)	Total	7	10	**19.5**	14
	Average	0.175	0.250	**0.4875**	0.350
Hard Scores (JSON)	Total	4	7.5	**14**	12
	Average	0.10	0.1875	**0.35**	0.30
Easy Scores (Sentence)	Total	1	6	6	**9**
	Average	0.05	0.30	0.30	**0.45**
Medium Scores (Sentence)	Total	2.5	**16**	8	13.5
	Average	0.0625	**0.40**	0.20	0.3375
Hard Scores (Sentence)	Total	1	**13**	4	**13**
	Average	0.025	**0.325**	0.10	**0.325**
Total Scores (JSON)	Total	19	22	**45.5**	35.5
	Average	0.19	0.22	**0.455**	0.355
Total Scores (Sentence)	Total	4.5	35	18	**35.5**
	Average	0.045	0.35	0.18	**0.355**

5.2 Choosing the RL Algorithm

Proximal Policy Optimization (PPO) and Deep Q-Network (DQN) were the policy-based and value-based RL algorithms used for this study.

To compare their performance, many runs were evaluated with different hyperparameters. The possible values for the various hyperparameters were based on those in existing work in CybORG and other PPO and DQN implementations [77, 76, 79, 7, 6]. In particular, 162 iterations were trialed, with PPO consisting of combinations of the following:

- Batch size: $8, 16$
- Training interval: $16, 32, 64$
- Critic Learning Rate: $e^{-4}, 5e^{-4}, e^{-3}$
- Actor Learning Rate: $e^{-4}, 5e^{-4}, e^{-3}$
- Policy Clip: $0.1, 0.15, 0.2$

486 iterations were trialed for DQN, using combinations of:

- Batch size: $8, 16$
- Queue size: $100, 200, 300$
- Learning rate: $e^{-4}, 5e^{-4}, e^{-3}$
- Discount factor: $0.85, 0.9, 0.95$
- Epsilon: $0.9, 0.95, 0.99$

- Epsilon decay rate: $0.98, 0.99, 0.995$

The reason that DQN has approximately three times as many iterations is that it includes a larger number of hyperparameters. As a result, more iterations were required to evaluate all possible combinations.

The mean of the last 10 episodes over five iterations was used as the metric to determine the best-performing model. 150 episodes were run for each iteration. These tests were done in both a simpler 4-host network and the standard 13-host network.

The results of the comparison, along with their associated hyperparameters, are shown in Table 5.5 and Table 5.6. To keep the tables concise, only the best 10 combinations are shown for DQN and PPO.

Table 5.5: PPO hyperparameters and corresponding mean reward in a simulated 13-host and 4-host network. Only the top 10 results are shown, ordered by their respective mean reward.

Env	Batch Size	Train Int.	Critic LR	Actor LR	Policy Clip	Mean Reward
13 Host	16	16	0.0010	0.0010	0.15	-48.484
	8	16	0.0005	0.0010	0.10	-50.278
	8	32	0.0010	0.0005	0.15	-53.390
	8	32	0.0010	0.0005	0.10	-54.738
	16	64	0.0010	0.0010	0.20	-55.788
	16	32	0.0005	0.0005	0.10	-56.820
	8	32	0.0010	0.0005	0.20	-58.260
	16	32	0.0010	0.0010	0.10	-60.694
	8	32	0.0001	0.0010	0.20	-61.328
	8	16	0.0010	0.0010	0.15	-61.982
4 Host	8	32	0.0010	0.0010	0.15	-4.380
	16	16	0.0010	0.0005	0.15	-5.380
	16	32	0.0010	0.0005	0.10	-5.549
	8	32	0.0010	0.0010	0.20	-6.150
	8	16	0.0010	0.0001	0.10	-6.300
	16	32	0.0005	0.0005	0.15	-6.370
	16	16	0.0010	0.0005	0.10	-9.100
	8	64	0.0005	0.0005	0.10	-9.730
	8	32	0.0010	0.0005	0.20	-10.060
	8	16	0.0005	0.0001	0.15	-10.440

Table 5.6: DQN Hyperparameters and corresponding mean reward in a simulated 13-host and 4-host network. Only the top 10 results are shown, ordered by their respective mean reward.

Env	Batch Size	Queue Size	LR	Discount	Epsilon	Decay	Mean Reward
13 Host	8	300	0.0010	0.90	0.95	0.995	-107.74
	8	200	0.0010	0.90	0.95	0.995	-136.00
	8	300	0.0010	0.85	0.95	0.99	-141.28
	8	300	0.0010	0.85	0.90	0.99	-142.96
	16	300	0.0001	0.95	0.99	0.995	-144.35
	16	300	0.0001	0.95	0.90	0.99	-147.01
	8	100	0.0005	0.85	0.90	0.99	-151.50
	16	300	0.0010	0.95	0.90	0.995	-153.17
	8	300	0.0010	0.85	0.99	0.99	-157.03
	8	100	0.0005	0.95	0.95	0.99	-157.59
4 Host	8	200	0.0010	0.85	0.90	0.99	-5.37
	8	300	0.0005	0.90	0.95	0.99	-5.77
	8	100	0.0010	0.85	0.99	0.99	-5.91
	8	200	0.0010	0.85	0.99	0.98	-5.98
	8	200	0.0010	0.85	0.99	0.99	-5.99
	8	300	0.0010	0.85	0.95	0.98	-6.12
	8	200	0.0010	0.95	0.99	0.98	-6.14
	8	300	0.0005	0.85	0.95	0.99	-6.20
	8	200	0.0005	0.85	0.95	0.99	-6.26
	8	300	0.0010	0.90	0.90	0.99	-6.26

As shown in the above tables, the PPO algorithm consistently yielded higher rewards than DQN despite having less than half the iterations. Furthermore, after manually analyzing the plots for both algorithms, PPO consistently demonstrated smoother convergence to a policy.

Figure 5.3 shows the plots of the best performing DQN implementation against the best performing PPO implementation for the 13-host simulated network. Because of the superior performance demonstrated by the PPO agent, it is the RL algorithm used for this research.

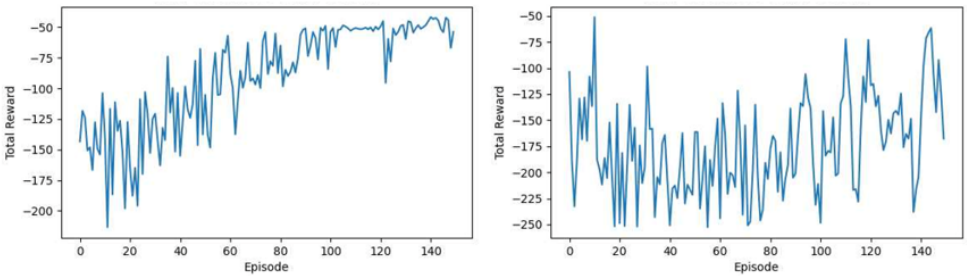


Figure 5.3: Comparing the performance of a PPO agent (left) and a DQN agent (right) based on environment rewards. Results show the mean over five runs with 150 episodes per run. It should be noted that these plots are on different y-scales as the intention is to compare general convergence trends.

The initial randomness exhibited by the DQN algorithm in Figure 5.3 is expected due to the nature of using epsilon to balance exploration with exploitation. For this implementation, it starts with a 95% chance of selecting a random action, which is decayed by 0.05% every batch. However, by episode 100, there is a 12.79% chance of selecting a random action, and by episode 150, there is a 4.69% chance. This makes the continued stochasticity and lack of convergence not solely attributable to the epsilon-greedy strategy implemented.

5.3 Environment Modifications

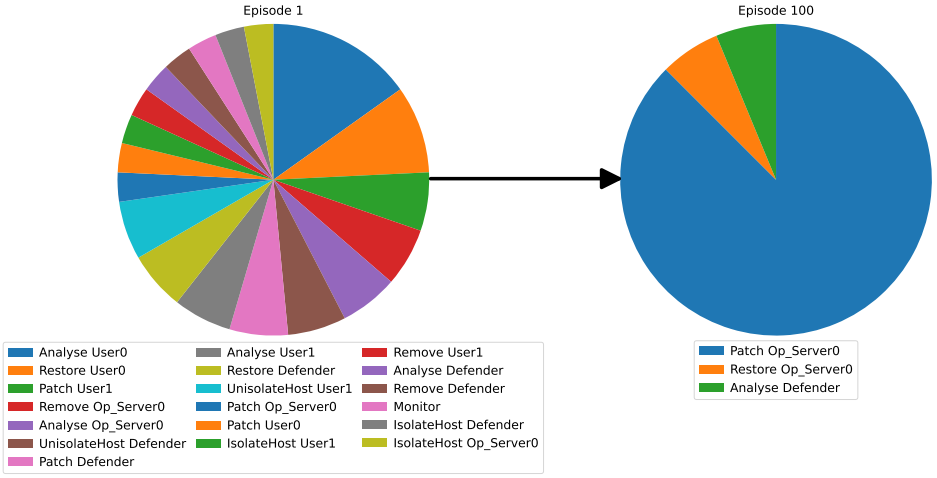
5.3.1 Adding Actions

It is important that actions are fine-tuned to ensure that there is a balance between being consistently exploited and never chosen. For example, with the patch action, it is important that it decreases the likelihood of a host being exploited, but does not completely eliminate the possibility of the red agent gaining a foothold, and progressing through the unified kill chain.

Figure 5.4 illustrates the fine-tuning process for the patch action. The top plot shows the patch action decrease the chance of an exploit succeeding by 20% with the red agent increasing the chance of success by 20%. It can be seen that the blue agent ends up repeatedly patching the operational server, making it impossible for the attacker to exploit.

When the parameters are adjusted to the blue agent decreasing the chance of a successful exploit by 30% and the red agent increasing it by 35%, blue stops only exploiting patch for a single host, since red has a bigger impact.

(a) Patch of 0.2 for Blue and Red



(b) Patch of 0.3 for Blue and 0.35 for Red

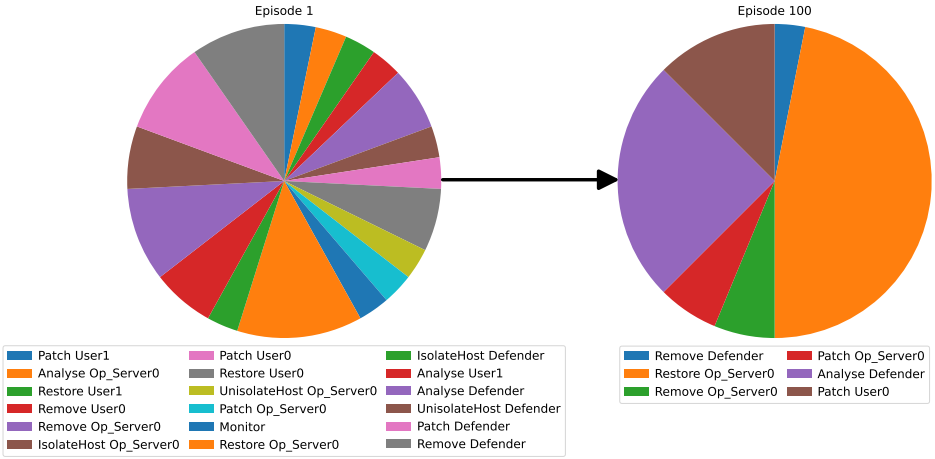


Figure 5.4: Comparing hyperparameters for the patch action on episode 1 (left) and episode 100 (right). The top configuration shows the blue agent's patch action decreasing likelihood of the exploit succeeding by 20% and the red agent's exploit action increasing the likelihood of exploit success by 20%. The bottom configuration shows blue decreasing the likelihood by 30% and red increasing it by 35%.

A similar process was applied for the isolate and block actions. Patch required the most tuning to make it effective, but not dominate over the other actions. For the patch action, the configuration was as follows:

- Performing the patch action: increases the host's patch score by 30%.

- Performing the exploit or privilege escalate action: decreases the host's patch score by 35%.

For the block action:

- Each timestep a low-valued host is blocked: -0.1 to the reward.
- Each timestep a medium-valued host is blocked: -0.2 to the reward.
- Each timestep a high-valued host is blocked: -0.3 to the reward.

For the isolate action:

- Each timestep a low-valued host is isolated: -0.2 to the reward.
- Each timestep a medium-valued host is isolated: -0.4 to the reward.
- Each timestep a high-valued host is isolated: -0.5 to the reward.

Removal of the Block Action

The block action was excluded from this study due to its inconsistent input size compared to the rest of the actions. Unlike the other actions, which operate on a single host, the block action requires two hosts. This difference introduced complications when integrating the LLM - it would either output a single host for the block action, or two hosts for the other actions. Basic prompt engineering was insufficient to consistently resolve this problem, so the block action was omitted from the action space.

5.3.2 Signal Modifications

As discussed, the feature space and the reward signals were modified. The feature space was changed from having 7 bits to represent each host to 4 floats between 0 and 1, with 2 additional floats representing the total number of compromised and isolated hosts.

Additionally, the reward signal was changed to be normalized between -2.5 and +2.5 rather than always being negative. This was to enable the PPO agent to better distinguish between good and bad actions.

The difference between the new and old signals using the same PPO agent is illustrated in Figure 5.5. The updated signals show slightly improved convergence with respect to stability and yield roughly the same results.

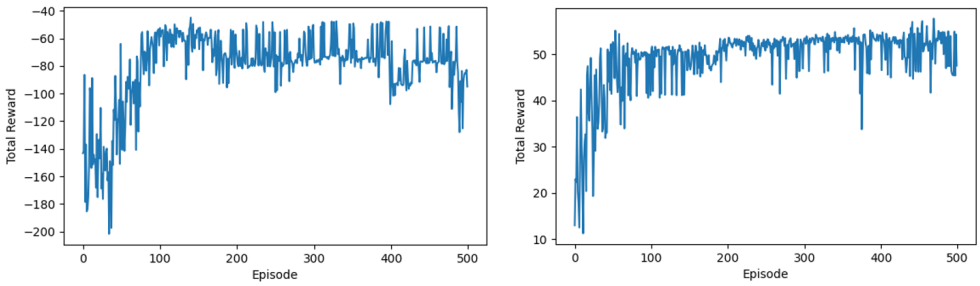


Figure 5.5: Comparing the performance of learning using the original reward schema and feature space (left), to learning with normalized rewards and the updated feature space (right). Results are averaged over 10 independent runs with 500 episodes each. Kept as two separate graphs due to the difference in the y-scale.

Figure 5.5 shows that the agent converges to ≈-70 when using the old configuration, and ≈54 when using the updated reward signals and feature space.

Converting these into the average reward per timestep becomes:

$$\frac{-180}{32} = \text{-5.625 and } \frac{54}{32} = 1.6875$$

where 32 is the number of timesteps in an episode.

Normalizing the former between -2.5 and 2.5:

$$\frac{-2.1875 + 13.1}{13.1} * 5 - 2.5 = 1.665$$

where 13.1 is the magnitude of the lowest reward for a single timestep in the 13-host scenario (0 is the highest score for a single timestep, and -13.1 is the lowest).

This small difference between a reward of 1.6875 and 1.665 is not sufficient evidence to claim the benefits of modifying the feature space and normalizing the reward signal; however, Figure 5.5, illustrates a superior convergence trend with the modified rewards. There are no apparent dips in performance that are present under the old configuration (e.g., the dip at ≈episode 400).

5.4 Optimizing PPO

The PPO agent's configuration showed quick improvement and convergence; however, after further analysis, it appeared to converge to a local minimum and exhibited some unstable behavior. In particular:

- The episode reward quickly converges and plateaus at roughly episode 100; and
- The explained variance, a metric measuring how good the critic is at estimating returns [87] decreases to a negative amount; and
- The critic quickly converges to 0 at around the same time rewards plateau - an indication of early over-fitting; and
- The entropy quickly converges to almost 0 at roughly the same time, signifying a highly deterministic policy very early in training.

One of the main reasons for this behavior appears to be the low number of samples used for each training interval, resulting in the models overfitting to a subset of the possible states. Furthermore, comparing the PPO agent's implementation to Stable Baseline3's [78], certain optional components are missing, such as the addition of entropy to the loss signal and gradient clipping for both the critic and actor. Gradient clipping sets a threshold on the steepness of gradients during backpropagation, ensuring that parameter updates are contained, which contributes to overall stability.

To rectify these potential deficiencies, Optuna [84], an open-source Bayesian optimization framework, was used for additional hyperparameter tuning. In particular, it was configured to discover the best combinations with respect to policy convergence using the following parameters:

- Batch size of 64, 128 and 256, timesteps
- Training interval size of 256, 512, and 1024 timesteps
- Critic and actor learning rates between $1e^{-}5$ to $3e^{-}3$
- Epochs between 10 and 30
- Critic gradient clip between 0.1 and 0.5
- Policy gradient clip between 0.1 and 0.5
- Entropy coefficient between 0.0001 and 0.01
- Entropy decay between 0.95 and 0.9995

Three hundred trials were conducted where each trial consisted of the average of 300 episodes over 10 independent runs. The success criteria for tuning was the average reward over the last 20 episodes.

Unlike the previous hyperparameter tuning for comparing PPO with DQN, the chosen configuration was not based solely on the performance of Optuna. Manual adjustments were made to the parameters in an attempt to achieve the most stable implementation. The difference in performance between the old PPO implementation and the updated one is shown in Figure 5.6.

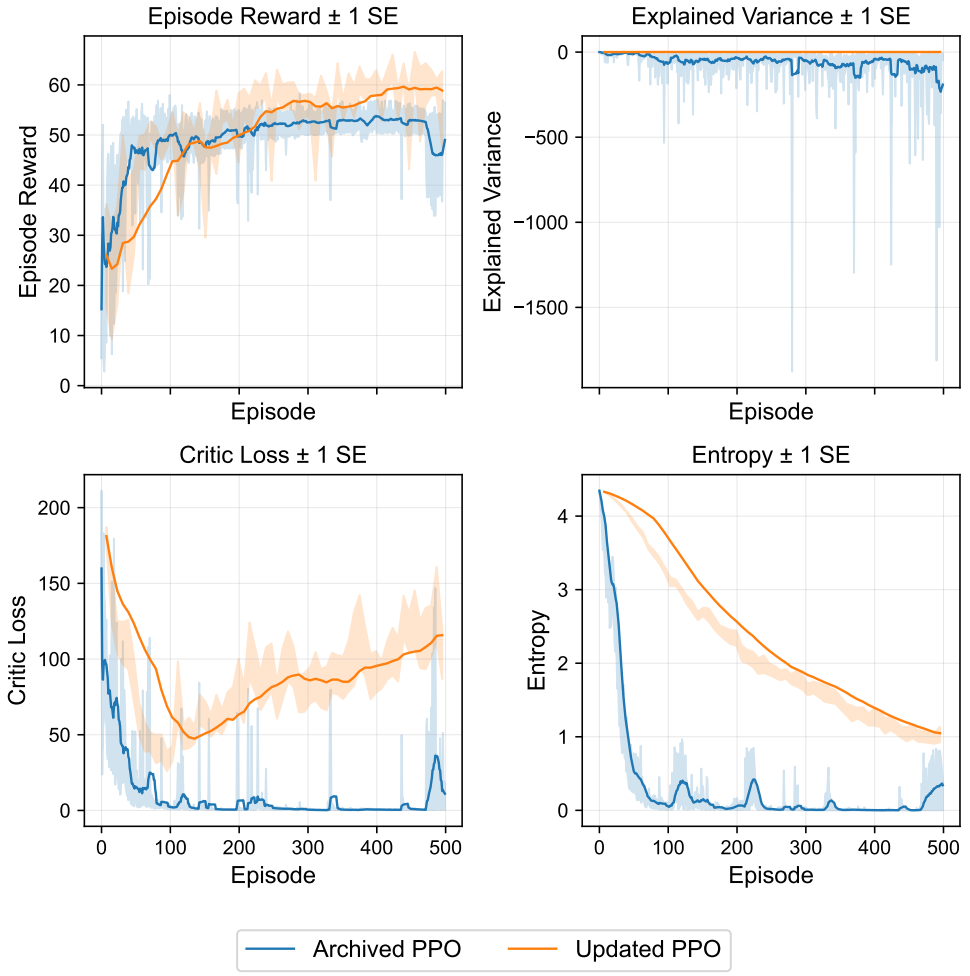


Figure 5.6: Comparing the performance of the old PPO implementation with the new one across episode rewards, explained variance, critic loss, and entropy. These results are the mean of 10 independent runs across 500 episodes, with a standard error of ± 1.

As illustrated in Figure 5.6, the new PPO implementation converges to a higher reward. Furthermore, its explained variance remains positive, with a steadily decreasing entropy as the agent prioritizes exploitation over exploration. Similarly, the critic loss does not overfit and converge to zero abruptly, but actually increases. This is concerning in traditional supervised applications, where the goals is to minimize loss across fixed targets; however, in PPO - and RL in general - the target is a function of the actor network, which is also improving. Because of this moving target, the observed increase in the critic's

loss signals is not necessarily indicative of a poorly performing agent.

Table 5.7 summarizes the differences in hyperparameters between the old and new PPO implementations.

Table 5.7: Comparison of hyperparameters between the old and new PPO agents. "N/A" indicates that particular parameter was not applied.

Hyperparameter	**Old PPO Agent**	**New PPO Agent**
Episode Size	32	32
Batch Size	16	256
Training Interval Size	16	256
Critic LR	0.001	0.0016
Policy LR	0.001	0.0016
Epochs	30	30
Policy Clip	0.15	0.2
Entropy Coef	0	0.005
Entropy Decay	0	0.99
Critic Grad Clip	N/A	0.1
Policy Grad Clip	N/A	0.5

5.5 Comparing Teacher-Guided Methods

This section evaluates the performance of the various teacher-guided implementations against the baseline PPO agent. These techniques include action masking, feature space modification, reward shaping, auxiliary loss, and combinations thereof.

The purpose of this evaluation is to determine the most effective teacher-guided algorithm with respect to training efficiency and learning stability. The most effective algorithm is then used to incorporate the LLM into the RL pipeline. The main metric used for the evaluation is the rewards outputted by the CybORG environment.

5.5.1 Action Masking

Action masking integrates the teacher into the RL pipeline by having it modify the actor's probability distribution, increasing the likelihood of its recommended action being sampled.

Different implementations were attempted with action masking. In particular:

- Softmax masking for actions; and
- Softmax masking for hosts; and

- Logit masking for actions.

Different methods were explored within these implementations, including:

- Inference-only, where the masking is only applied during the agent's data collection phase, and the ratio between the original and new probability distribution during training remains unchanged.
- New probabilities, where the masking is applied at inference and to the new probability distribution during training. The purpose of this is to encourage the agent to converge onto the masked policy.
- New and old probabilities, where the masking is applied during inference, and to the old and new probability distributions during training. The purpose of this is to encourage the agent to converge onto the masked policy, while maintaining a stable ratio between the old and new policies.

Masking Actions via Softmax

One way action masking can be performed is to directly modify the actor's probability distribution to favor the teacher's recommendation. For these implementations, masking is applied based on the single recommendation from the teacher, reducing the probability of selecting any other action.

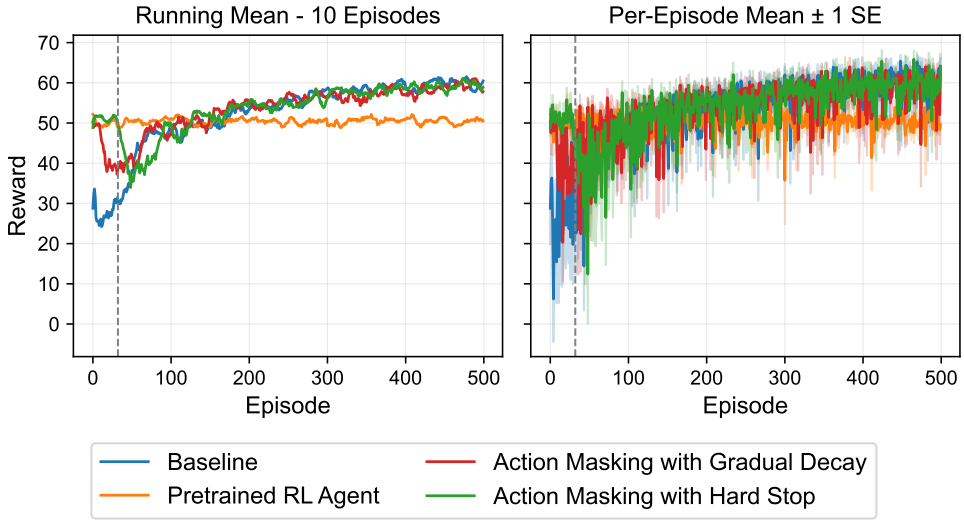
The three methods for softmax-based action-masking were implemented and the results are as follows:

- Inference only (Figure 5.7a); and
- New probabilities (Figure 5.7b); and
- New and old probabilities (Figure 5.7c).

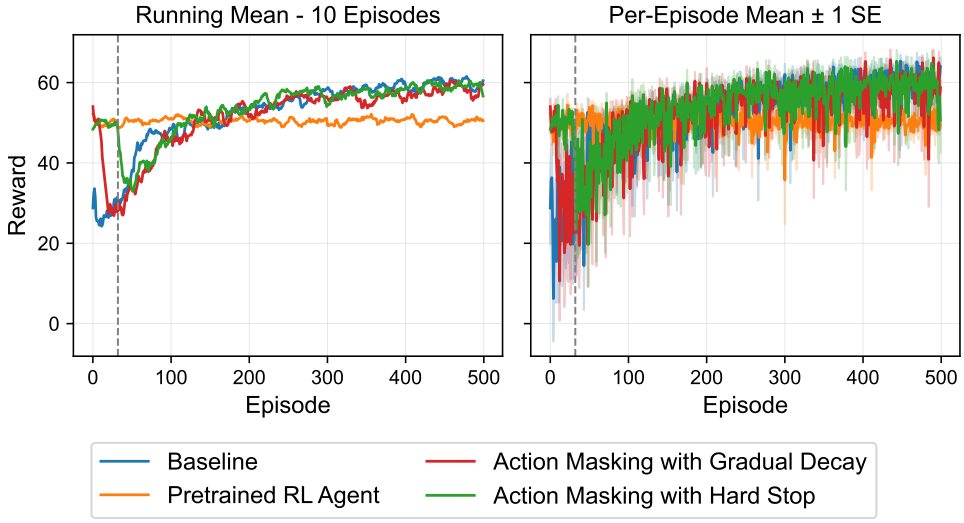
Two different implementations were tested for each technique: a gradual decay, where the teacher's influence is reduced incrementally, and a hard stop, where the teacher's impact is completely removed. The intention of the gradual decay was to facilitate a smoother transition to independent RL. In contrast, the hard stop was meant to create a clear separation between the teacher-guided and independent RL phases, enabling an evaluation of policy robustness and the potential benefits of a steeper transition. The two implementations were the same across all algorithms:

- The gradual decay reduced the impact of masking by 25% every training interval (eight episodes); and
- The hard stop fully removed the masking after four intervals (32 episodes).

(a) Inference-Only Masking



(b) Inference with New-Policy Masking



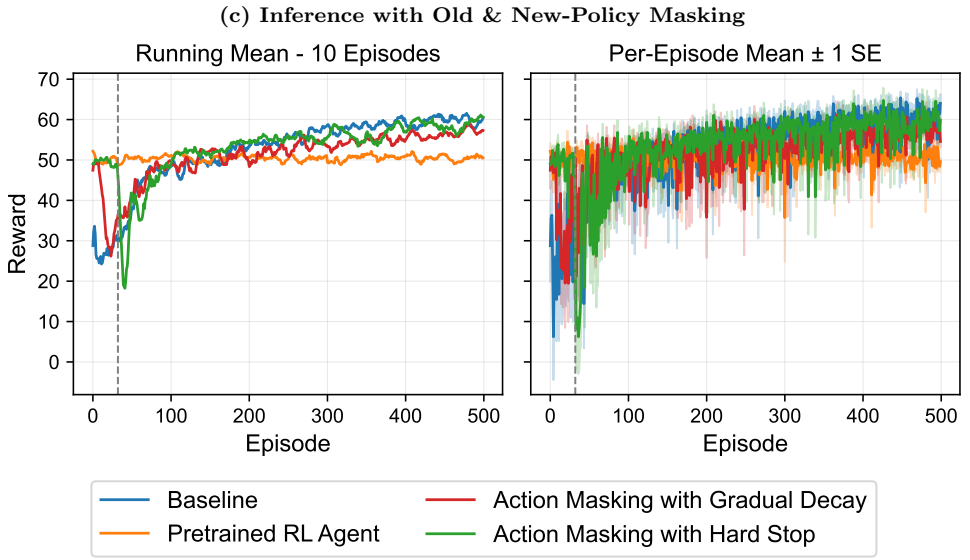


Figure 5.7: Masking actions via softmax modification against the PPO baseline for 10 independent runs. The vertical dashed line indicates the point at which the teacher guidance is abruptly removed for the hard stop configuration (the green curve).
Left: 10-episode running mean.
Right: Per-episode mean reward with ±1 standard-error.

Figures 5.7a-c demonstrate that masking actions has an initial positive impact on performance. For inference-only, it outperforms the baseline for the first ≈50 episodes, demonstrating the teacher's potential to eliminate the requirement of selecting obviously unfavorable actions to learn. All implementations show a noticeable drop in performance during the transition from teacher-guided to independent RL. The lowest of these drops is the the gradually decaying impact for inference-only masking (Figure 5.7a), and the highest is the abrupt stop for old and new policy masking, where performance dips lower than the baseline's at episode ≈40. (Figure 5.7c).

The gradually decaying mask for inference-only shows the best results in terms of having the smallest dip in performance (episode reward of ≈50 to ≈40) and the most stable transition to independent RL, where at no point does it drops below the baseline agent's performance.

This demonstrates that gradually decaying the effect of the masking for inference-only facilitates a smoother and quicker transition to independent learning from the environment, converging to stable behavior approximately

50 episodes earlier than the hard stop implementation. While masking actions via softmax modification typically shows a positive impact on the early stages of training, it does not result in earlier convergence on an optimal policy.

Masking Hosts via Softmax

Similar to the above, this implementation incorporates the teacher's feedback by directly modifying the softmax; however, it alters the probability distribution based on a set of actions instead of a single one. In particular, the teacher recommends a host, and the probability of selecting any action not pertaining to that host is decreased. The intention here is to give the agent more freedom to explore actions, potentially outperforming the teacher during the guided phases.

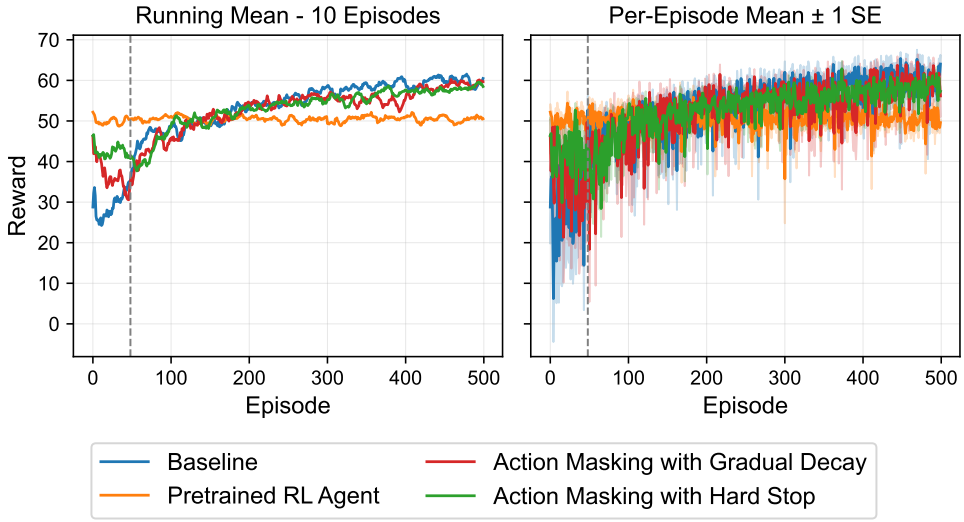
The same three methods were implemented, but applied to hosts instead of actions.

- Inference-only masking (Figure 5.8a); and
- Inference-only masking and masking the new probabilities during training (Figure 5.8b); and
- Inference-only masking and masking both the new and old probabilities during training (Figure 5.8c).

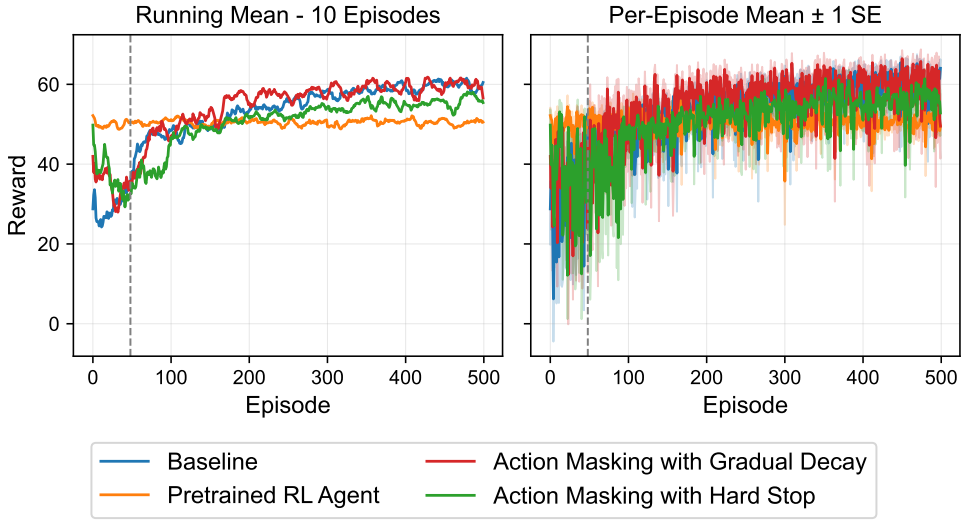
The gradual decay and hard stop were the same across the three algorithms:

- Gradual decay of 10% every training interval (eight episodes); and
- Complete removal of LLM-guidance after six training intervals (48 episodes) with no decay (hard stop).

(a) Inference-Only Masking



(b) Inference with New-Policy Masking



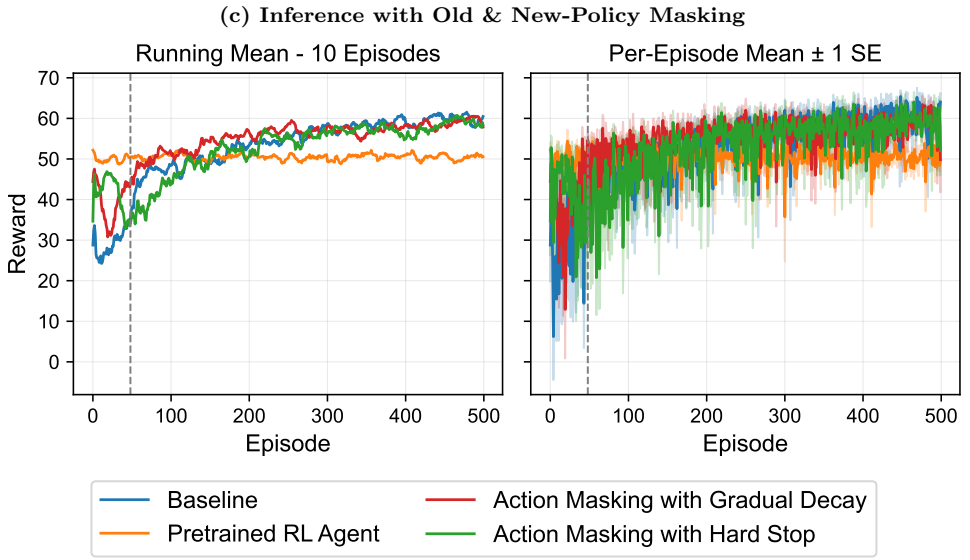


Figure 5.8: Masking hosts via softmax modification against the PPO baseline for 10 independent runs. The vertical dashed line indicates the point at which the teacher guidance is abruptly removed for the hard stop configuration (the green curve).
Left: 10-episode running mean.
Right: Per-episode mean reward with ±1 standard-error.

Figures 5.8a-c demonstrate a noticeable impact on initial training compared to the baseline PPO agent; however, this impact is less pronounced than that of masking actions. Inference-only with an abrupt transition from teacher-guided to independent RL (ref Figure 5.8a) shows the best initial performance and transition to independent RL; however, it falls short of the action masking technique described above. It does have a smaller dip during the transition to independent RL; however, this is relative to its lower initial performance - it drops to ≈30 during the transition whereas masking based on a recommended action only drops to ≈40.

Overall, masking hosts and enabling the RL agent to explore within this masked subset does not yield much benefit with respect to initial training efficiency compared to masking actions.

Masking Actions via Logits

Another way to incorporate the teacher's guidance with masking is to modify the probability distribution at the logit level before the softmax activation

function is applied. This was only evaluated using the masked new and old probabilities configuration. This is because masking only at inference changes the action that is sampled from the distribution - it does not matter how this occurs, whether through the modification of logits or softmax. The effect of masking actions through logit modification is illustrated in Figure 5.9.

The same hard and gradual decay approaches were taken with:

- The hard approach added a value of -500 to the logits corresponding to non-recommended actions for the first six training intervals (48 episodes).
- The decay started after one training interval and added -1, decreasing in magnitude by 0.2 every interval (-1 for interval two, -0.8 for interval three, etc).

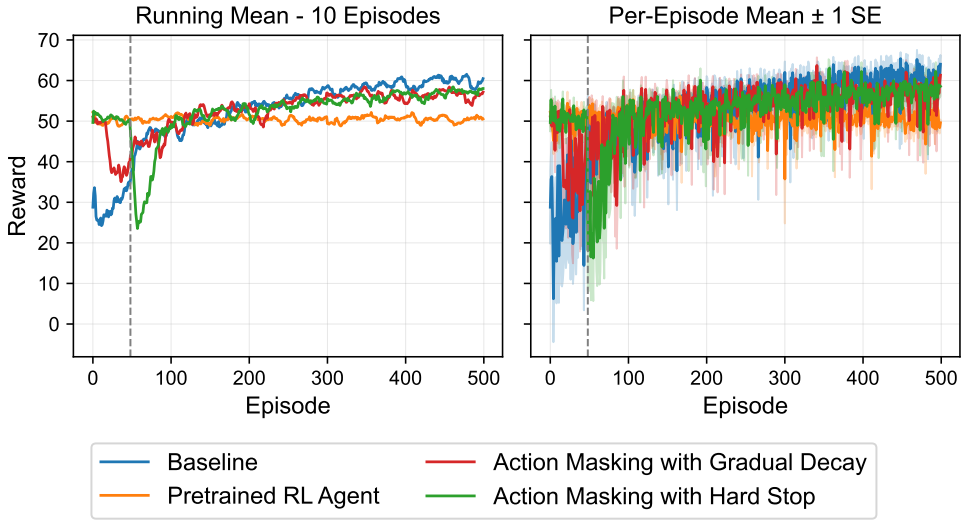


Figure 5.9: Comparison of masking actions via logit modification against the PPO baseline across 10 independent runs. The vertical dashed line indicates the point at which the teacher guidance is abruptly removed for the hard stop configuration (the green curve).
Left: Mean reward after applying a 10-episode running average.
Right: Per-episode mean with a ±1 standard error.

Figure 5.9 shows no noticeable improvement over the baseline PPO performance, with the hard stop implementation dropping below the baseline agent's performance at ≈episode 45.

Comparing Masking Techniques

Overall, action-masking shows promise in its demonstration of superior performance in early training compared to the PPO baseline; however, it exhibits noticeable drops during the transition from teacher-guided to independent RL. Masking actions during sampling (inference-only) showed the best results across all masking techniques in terms of initial performance and transition stability.

It should be noted that the hyperparameter tuning for these techniques was not as extensive as in previous stages of this research; however, minimal tuning did occur. For example, Figure 5.10 shows the combination of the masking with a gradual decay approach, where actions are completely masked until episode 48 and decayed by 25% per training interval thereafter.

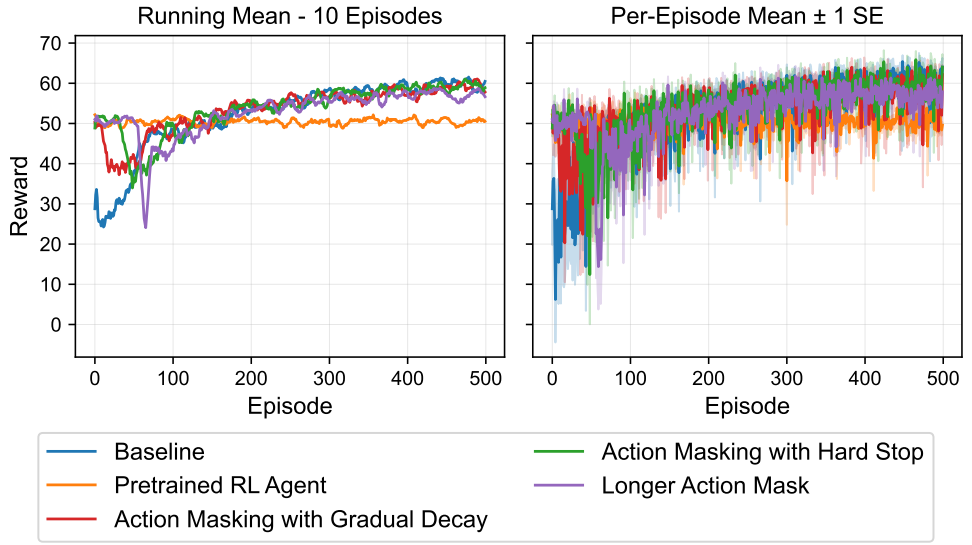


Figure 5.10: Illustrating the effects of an extended teacher-guided action masking phase before transitioning to independent RL.
Left: Mean reward after applying a 10-episode running average.
Right: Per-episode mean with a ±1 standard error.

Figure 5.10 shows that if the masking is applied for too long, it ultimately decreases the agent's performance, requiring more time to become stable and align with the baseline's performance. This illustrates the importance of balancing the exploitation of the teacher with the exploration of the environment for all of these teacher-guided implementations.

5.5.2 Feature Space Modification

Unlike action masking, which directly impacts the agent's distribution, the intent of feature space modification is to incorporate the teacher by providing the agent with additional information to make a decision. This additional information takes the form of a recommended action that is appended to its feature space. Figure 5.11 shows the performance of three different ways the teacher's recommendation can be appended to the agent's state space:

- As a binary value; and
- As a one hot encoded value; and
- As a float.

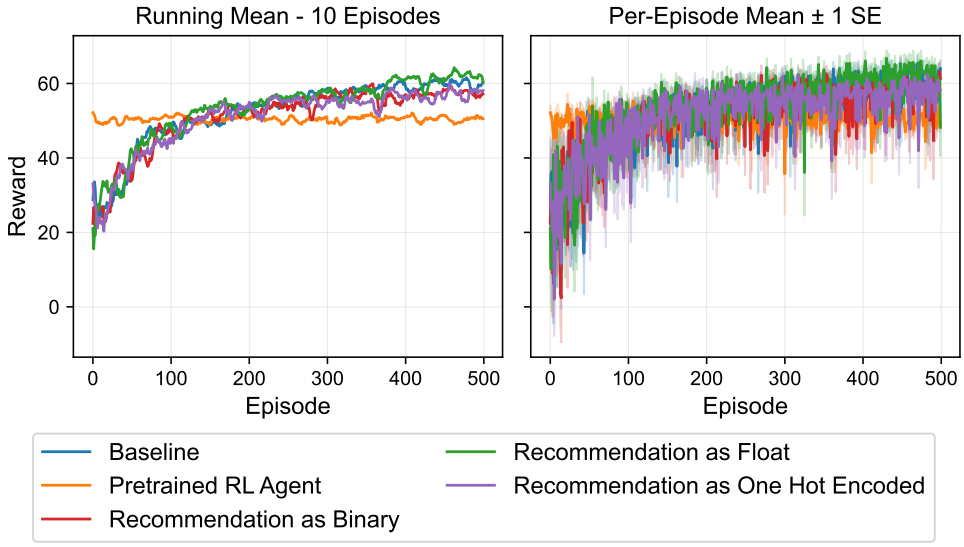


Figure 5.11: Comparison of Feature Space Modification against the PPO baseline across 10 independent runs.
Left: Mean reward after applying a 10-episode running average.
Right: Per-episode mean with a ±1 standard error.

Figure 5.11 demonstrates that there is no noticeable improvement in performance with any of the various feature space implementations.

Local Interpretable Model-agnostic Explanations (LIME) was then used to verify the impact that the teacher's features had on the agent's decision. Tables 5.8, 5.9 and 5.10 show the LIME results for the recommendation being appended as a float, one-hot encoded, and as a binary value respectively. The *Reco in Top 4* column indicates whether the teacher's recommendation is in the RL agent's top 4 most likely actions to choose. The Direction column

indicates whether the feature pushed the RL agent towards its policy or away from it. The Ranking column indicates how strong the teacher's feature is with respect to every other feature. The number of features for each implementation is:

- 55 total features for the recommendation as a float; and
- 132 total features for the one-hot encoded recommendation; and
- 61 total features for the recommendation as a binary.

The LIME results for the one-hot encoded implementation only illustrate the impact of the one-hot encoded feature (the feature that is 1) on the agent's decision.

The same state described in Figure 4.20 was used across all implementations to produce the LIME results.

Table 5.8: LIME results for appending the LLM recommendation to the feature space as a float. Reco in Top 4 shows if the teacher's recommendation is in the top 4 actions from the RL agent's policy, with its associated ranking if it is.

Episode	**Weight**	**Max Weight**	**Ranking**	**Direction**	**Reco in Top 4**
1	-5.58E-06	8.54E-05	40	Away	No
8	-1.46E-04	2.63E-03	33	Away	No
16	-1.32E-03	8.10E-03	25	Away	No
50	-2.14E-04	1.31E-02	44	Away	No
100	-2.03E-03	-6.65E-02	44	Away	No
200	1.00E-02	-2.04E-02	40	Towards	No
300	-1.51E-02	2.12E-01	35	Away	No
500	-7.62E-02	2.87E-01	20	Away	Yes/1

Table 5.9: LIME results from appending the teacher's recommendation to the feature space as a a one-hot encoding. Only the weights for the one-hot encoded feature are included. Reco in Top 4 shows if the teacher's recommendation is in the top 4 actions from the RL agent's policy, with its associated ranking if it is.

Episode	**Weight**	**Max Weight**	**Ranking**	**Direction**	**Reco in Top 4**
1	1.22E-05	-8.39E-05	36	Towards	No
8	-6.90E-05	1.48E-03	71	Away	No
16	2.78E-03	-1.06E-02	30	Towards	No
50	6.56E-03	-3.22E-02	18	Towards	No
100	2.66E-02	-9.29E-02	24	Towards	No
200	4.84E-02	-2.45E-01	38	Towards	No
300	3.13E-01	-3.99E-01	3	Towards	No
500	2.49E-01	3.31E-01	2	Towards	No

Table 5.10: LIME results for appending the teacher's recommendation to the feature space as a binary vector. Due to the large number of columns, headers are truncated and numbers are rounded to one decimal place.

Ep	Feat0	Feat1	Feat2	Feat3	Feat4	Feat5	Feat6	Max Wt	Top 4
1	-2.5e-5 (29)	**-1.1e-4 (3)**	**1.7e-4 (1)**	**8.6e-5 (7)**	-1.3e-6 (52)	**-1.0e-4 (5)**	**1.0e-4 (6)**	1.7e-4	No
8	**-1.1e-3 (8)**	5.7e-5 (47)	-8.3e-4 (13)	-2.9e-4 (25)	-3.3e-4 (22)	1.5e-4 (36)	**1.1e-3 (9)**	3.2e-3	No
16	**3.4e-3** (3)	**-4.0e-3 (1)**	**2.0e-3 (8)**	4.1e-4 (31)	**2.2e-3 (7)**	**1.9e-3 (9)**	-1.6e-4 (43)	-4.0e-3	No
50	**-3.2e-2 (4)**	9.8e-3 (27)	-1.0e-2 (25)	1.6e-2 (15)	6.2e-3 (33)	**-4.5e-2 (1)**	1.6e-2 (16)	-4.5e-2	No
100	**1.4e-2 (8)**	**-1.5e-2 (6)**	**1.9e-2 (4)**	8.2e-3 (20)	-7.2e-3 (22)	**2.9e-2 (3)**	1.7e-3 (40)	3.3e-2	No
200	**-1.5e-1 (1)**	2.8e-2 (19)	**-9.0e-2 (3)**	-1.1e-2 (36)	**-6.4e-2 (10)**	**8.4e-2 (5)**	**6.6e-2 (9)**	-1.5e-1	No
300	-5.3e-3 (37)	**3.5e-2 (9)**	**3.9e-2 (7)**	-3.3e-2 (11)	-1.5e-2 (19)	**5.0e-2 (5)**	1.9e-2 (17)	3.3e-1	No
500	-2.7e-2 (26)	6.3e-2 (15)	5.9e-2 (16)	**-9.0e-2 (7)**	5.1e-2 (19)	**1.4e-1 (2)**	5.1e-2 (20)	2.5e-1	No

The results from Tables 5.8, 5.9 and 5.10 show that the agent never prioritizes the action recommended by the teacher based on its associated features. It can be seen that at episode 500 for mapping the teacher's action to a float (ref: Tab 5.8), the RL agent selects the recommendation; however, the feature meant to map that recommendation to an executable action is weighted at less than 5% of the most impactful feature and is ultimately driving the agent away from the recommendation.

The one-hot encoded configuration illustrates a general increase in the impact that the one-hot encoded feature has across episodes; however, it never maps this to a likely action in its policy. This could be due to the fact that the one-hot encoded feature is always 1, naturally having a higher impact on the gradient due to its larger value (since all features are normalized between 0 and 1). Overall, teacher-guided RL via feature-space modification on its own shows no noticeable improvement in training compared to the baseline PPO agent.

5.5.3 Reward Shaping

Reward shaping involves modifying the rewards to encourage the agent to mimic the teacher's behavior. Similar to the above approaches, two implementations were attempted:

- Gradually decreasing the magnitude of the reward for selecting an action recommended by the teacher. This was done by adding 2.5 during the first interval and decreasing it by 10% thereafter if the agent selected an action recommended by the LLM, and adding 1.0 during the first interval and decreasing it by 10% thereafter if the agent selected an action that pertains to the host recommended by the LLM.
- Abruptly ceasing the extra rewards at a certain interval, where 2.5 was added to actions that matched the teacher's recommendations and 1.0 was added to actions that contained the recommended host. These rewards were constant and abruptly halted at training interval five (episode 40).

The results of teacher-guided RL via reward shaping are shown in Figure 5.12. It should be noted that the rewards shown in the figure are from the original environment (i.e., exclude any additional teacher rewards).

Figure 5.12: Comparison of reward-shaping against the PPO baseline across 10 independent runs. The vertical dashed line indicates the point at which the teacher guidance is abruptly removed for the hard stop configuration (the green curve).
Left: Mean reward after applying a 10-episode running average.
Right: Per-episode mean with a ± 1 standard error.

Overall, Figure 5.12 shows no noticeable improvement from incorporating the teacher's feedback using reward shaping alone, whether the signal is gradually reduced or abruptly halted. In fact, reward shaping with an abrupt halt to the teacher-supplemented rewards demonstrates slightly worse performance with respect to later convergence; however, this could simply be due to the stochastic nature of the environment.

5.5.4 Auxiliary Loss

Similar to reward shaping, auxiliary loss incorporates the teacher feedback during training; however, its purpose is to directly modify the loss to better align with the teacher's recommendation, rather than through the modification of the reward signal.

Figure 5.13 demonstrates the results of having the teacher's impact on the agent's overall loss gradually decayed and abruptly stopped.

The decayed version had the teacher's influence decrease by 25% each training interval, starting at episode eight. During each decay, the entropy coefficient was increased by $5e^{-4}$ to encourage exploration during independent RL. Once in independent RL, the entropy coefficient was decreased by $2e^{-4}$ until it matched the baseline PPO's configuration.

The abruptly stopped implementation had the LLM's influence stop completely after three training intervals, with the entropy coefficient increasing by $5e^{-4}$ every teacher-guided interval, then decreasing by $2e^{-4}$ every interval until it reached the baseline configuration.

Figure 5.13: Comparison of Auxiliary Loss against the PPO baseline across 10 independent runs. The vertical dashed line indicates the point at which the teacher guidance is abruptly removed for the hard stop configuration (the green curve).
Left: Mean reward after applying a 10-episode running average.
Right: Per-episode mean with a ± 1 standard error.

As illustrated in Figure 5.13, there are considerable improvements in initial convergence to the teacher's policy. For both implementations, they plateau after quickly converging onto the teacher's policy and then increase at roughly the same rate as the baseline performance. Gradually decaying the impact of the auxiliary loss shows slightly improved final performance; however, this could be due to the stochastic nature of the environment.

5.5.5 Combining Implementations

After evaluating each of the teacher-guided implementations separately, combinations were tested in the hopes of increasing training efficiency.

Reward Shaping and Feature Space Modification

The results for incorporating reward shaping and feature space modification on their own demonstrated no noticeable gain with respect to training efficiency. The idea behind combining these two approaches is to give the agent an incentive for selecting the teacher's recommendation. If the agent is rewarded for selecting the teacher's recommendation, this should increase its ability to map that recommendation into a corresponding executable action.

The results for the combination of feature space modification and reward shaping are illustrated in Figure 5.14. Only one-hot encoded and float formats for the feature space are implemented due to the inherent difficulties of mapping a binary representation to a recommended action, shown in the previous LIME analysis.

The gradual decay implementation, with the same hyperparameters described above, is used for the reward shaping.

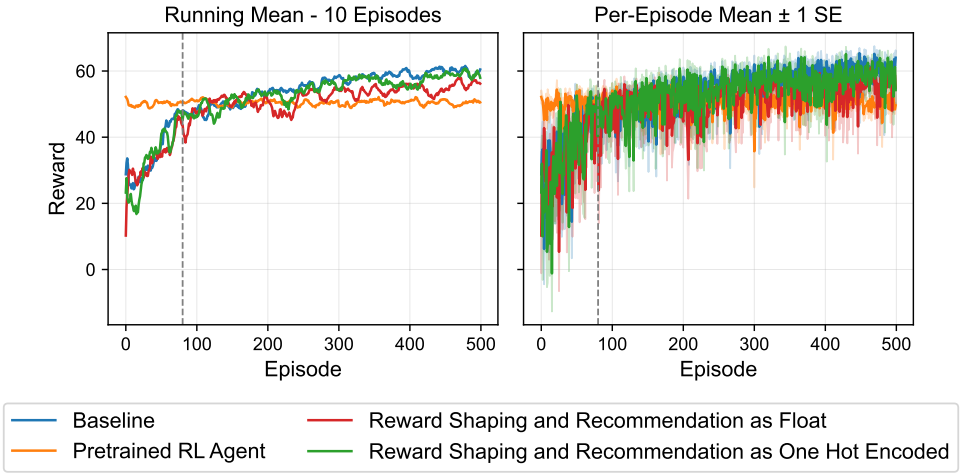


Figure 5.14: Comparison of reward shaping with feature space modification against the PPO baseline across 10 independent runs. The vertical dashed line indicates the point at which the teacher guidance has been completely removed after being gradually decayed by 10% starting at episode 8.
Left: Mean reward after applying a 10-episode running average.
Right: Per-episode mean with a ±1 standard error.

As shown in Figure 5.14, there is no noticeable improvement to training efficiency, and appending the recommendation as a float appears to decrease overall policy convergence. This is likely due to the randomness present within the environment; however, it could be due to the additional features deteriorating the agent's ability to make effective decisions based on its observed state.

The LIME analysis for the teacher's recommendation formatted as a float and as a one-hot encoding is shown in Tables 5.11 and 5.12, respectively.

Table 5.11: LIME results from appending the teacher's recommendation to the feature space as a float. Reco in Top 4 determines if the teacher's recommendation is in the top 4 actions from the RL agent's policy, with its associated ranking if it is.

Episode	**Weight**	**Max Weight**	**Ranking**	**Direction**	**Reco in Top 4**
1	8.98E-06	7.80E-05	31	Toward	No
8	3.21E-04	-4.06E-03	31	Away	No
16	-1.82E-04	1.28E-02	44	Away	No
50	2.86E-03	-5.72E-02	42	Towards	No
100	4.82E-04	-4.04E-02	44	Towards	No
200	-1.00E-03	3.81E-02	40	Away	No
300	2.59E-03	2.03E-01	45	Towards	Yes/1
500	2.97E-02	4.57E-01	34	Towards	Yes/1

Table 5.12: LIME results from appending the teacher's recommendation to the feature space as a a one-hot encoding. Shows weights for the one-hot encoded feature only. Reco in Top 4 determines if the teacher's recommendation is in the top 4 actions from the RL agent's policy, with its associated ranking if it is.

Episode	**Weight**	**Max Weight**	**Ranking**	**Direction**	**Reco in Top 4**
1	8.50E-06	2.64E-05	31	Towards	No
8	-7.27E-04	-2.05E-03	22	Away	No
16	4.14E-04	-7.43E-03	65	Towards	No
50	4.66E-04	1.85E-02	73	Towards	No
100	1.44E-02	-9.23E-02	30	Towards	No
200	1.14E-01	1.91E-01	4	Towards	No
300	2.22E-02	-3.18E-01	60	Towards	Yes/2
500	1.19E-01	3.75E-01	9	Towards	No

Similar to feature space modification on its own, Tables 5.11 and 5.12 show no correlation between the teacher's recommendation and the sampled action, illustrating a lack of ability to map the recommendation to an executable action. When the recommendation is mapped as a float or one-hot encoded value, the

agent does occasionally select the teacher's recommendation; however, it can be seen that the respective feature defining this recommendation has minimal impact on its choice.

Overall, combining feature space modification with reward shaping shows no noticeable improvement to training efficiency.

Action Masking and Feature Space Modification

Rather than providing an incentive to the RL agent to map its decisions to the teacher's recommendations through reward shaping, this idea is to force the agent to select particular actions, encouraging it to map these to the teacher's recommendations.

Inference-only action masking demonstrated the best performance out of the masking techniques with respect to training efficiency, so it was the chosen algorithm to be combined with feature space modification. Similar to reward shaping, the recommendation was appended as a float and a one-hot encoded value. For the action masking component, a mask was applied and gradually decayed by 25% every training interval, starting after episode 32.

The results of combining feature space modification with action masking are illustrated in Figure 5.15.

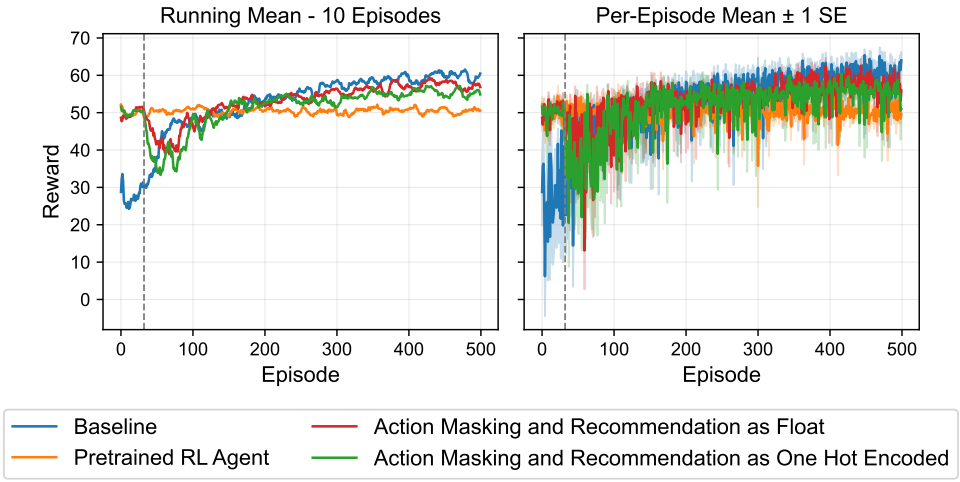


Figure 5.15: Comparison of feature space modification with action masking against the PPO baseline across 10 independent runs. The vertical dashed line indicates the point at which the teacher guidance begins to decay.
Left: Mean reward after applying a 10-episode running average.
Right: Per-episode mean with a ±1 standard error.

Figure 5.15 demonstrates the initial gains in performance provided by the action masking; however, it shows no noticeable improvement thereafter compared to action masking on its own. Furthermore, the policy convergence deteriorates slightly for both the float and one-hot encoded recommendations.

The LIME analysis for combining action masking with feature space modification for float and one-hot encoded recommendations is shown in Tables 5.13 and 5.14.

Table 5.13: LIME results from appending the LLM recommendation to the feature space as a float with action masking. Reco in Top 4 determines if the teacher's recommendation is in the top 4 actions from the RL agent's policy, with its associated ranking if it is.

Episode	**Weight**	**Max Weight**	**Ranking**	**Direction**	**Reco in Top 4**
1	-1.34E-06	8.48E-05	44	Away	No
8	-4.20E-04	-2.50E-03	18	Away	No
16	-1.09E-03	6.05E-03	26	Away	Yes/4
50	-1.49E-03	1.70E-02	40	Away	Yes/4
100	6.58E-05	3.90E-02	46	Towards	No
200	5.60E-03	4.78E-02	33	Towards	No
300	-1.30E-02	2.23E-01	35	Away	Yes/2
500	4.40E-03	3.10E-01	42	Towards	Yes/3

Table 5.14: LIME results from appending the LLM recommendation to the feature space as a a one-hot encoding with action masking. Shows weights for the one-hot encoded feature only. Reco in Top 4 determines if the teacher's recommendation is in the top 4 actions from the RL agent's policy, with its associated ranking if it is.

Episode	**Weight**	**Max Weight**	**Ranking**	**Direction**	**Reco in Top 4**
1	7.22E-07	-7.17E-05	82	Towards	Yes/3
8	1.62E-03	-2.79E-03	3	Towards	No
16	9.66E-04	-6.85E-03	46	Towards	No
50	2.40E-03	-1.37E-02	41	Towards	No
100	-1.30E-02	5.31E-02	27	Away	No
200	2.96E-02	2.86E-01	23	Towards	No
300	2.32E-02	-4.62E-01	52	Towards	No
500	-5.98E-03	1.52E-01	78	Away	No

Although appending the recommendation as a float shows the RL agent consistently selecting a probability distribution that includes the teacher's recommendation in its top four actions (ref Table 5.13), there is no noticeable correlation between the teacher's recommendation and the sampled action. The LIME analysis for one-hot encoding shows that the teacher's recommendation

is initially included in the top four actions; however, this can be ignored, as no learning has happened at this state, and the RL agent's weights are completely randomized.

It should also be noted that because the action masking is applied only at inference, the LIME results are for the RL agent's raw (unmasked) distribution.

Overall, combining action masking with feature space modification shows no greater improvement than action masking by itself, and the agent shows no indication of being able to map the teacher's recommendation to an executable action.

Action Masking and Auxiliary Loss

Masking actions at inference demonstrated initial gains in performance with a dip during the transition from teacher-guided to independent RL. On the other hand, auxiliary loss showed poor performance initially, but then quickly converged onto the teacher's policy. The purpose of combining action masking and auxiliary loss is to merge the quick performance gains that action masking offers while leveraging the auxiliary loss signal to facilitate a smoother transition to independent RL.

The gradually decaying action mask at inference, with the gradually decaying auxiliary loss was the chosen combination for this implementation, as they both yielded the highest results with respect to training efficiency. Both were decayed by 25% every training interval, starting after episode 8. The results of combining action masking with auxiliary loss are illustrated in Figure 5.16.

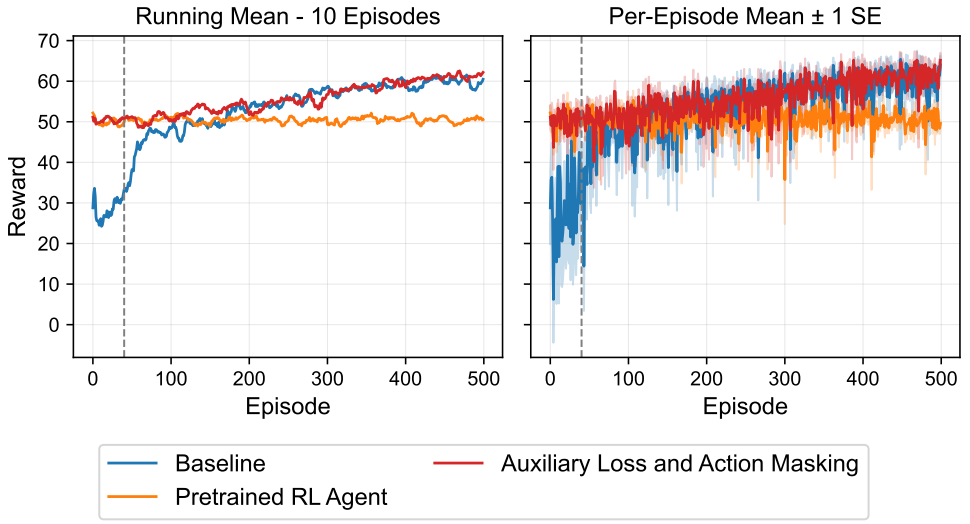


Figure 5.16: Comparison of action masking with auxiliary loss against the PPO baseline across 10 independent runs. The vertical dashed line indicates the point at which the teacher guidance is completely removed after having been decayed by 25% per training interval.
Left: Mean reward after applying a 10-episode running average.
Right: Per-episode mean with a ±1 standard error.

Figure 5.16 demonstrates the initial gain in training efficiency by incorporating action masking with auxiliary loss. The agent initially matches the teacher's performance without any noticeable drop in performance as it transitions to independent RL. It should also be noted that the agent's performance is noticeably superior following the transition to independent RL; however, the baseline catches up at episode ≈170.

5.5.6 Evaluating the Best Technique

Overall, incorporating the teacher's feedback using a combination of action masking and an auxiliary loss signal produced the best results with respect to training efficiency. While action masking and auxiliary loss demonstrate the potential to improve training, feature space modification and reward shaping had no noticeable impact.

It should be noted that certain techniques had a noticeably positive impact on initial performance and convergence; however, in every implementation, performance relative to the baseline was very similar by episode 500. The high-level comparison between the teacher-guided techniques is shown in Table

5.15.

Table 5.15: Comparison of teacher-guided techniques, using a pretrained RL agent as the teacher. Rank 1 yields the best performance with respect to training efficiency, whereas rank 4 is the least effective. All techniques that yielded no noticeable impact on training efficiency are ranked as 4.

Technique	**Best Configuration**	**Rank**	**Notes**
Action Masking with Auxiliary Loss	Gradually decaying mask at inference with gradually decaying teacher loss	1	High initial performance, with no drop during transition
Auxiliary Loss	Gradually decaying teacher loss	2	Quick convergence to teacher's policy
Action Masking	Masking actions only at inference with a gradually decaying mask	3	High initial performance, with drop during transition
Feature Space Modification	All configurations yielded similar performance	4	No noticeable improvement to training
Feature Space Modification with Action Masking	All configurations yielded similar performance	4	No noticeable improvement to training
Feature Space Modification with Reward Shaping	All configurations yielded similar performance	4	No noticeable improvement to training
Reward Shaping	All configurations yielded similar performance	4	No noticeable improvement to training

5.6 LLM Integration

After selecting the best performing teacher-guided algorithm with respect to training efficiency, the pretrained RL agent was replaced with the Cyber-Risk-Llama8B LLM [72].

5.6.1 Prompt Engineering

Before incorporating Cyber-Risk-Llama8B into the RL pipeline, the prompt was optimized further from what was used in the LLM selection process. This was mainly due to the unreliable and variable performance exhibited by executing the LLM's recommendations directly in CybORG.

The prompt engineering process was arguably the most time-consuming portion of this study due to its fundamental black-box nature. Generic prompt engineering techniques were employed; however, iterative progress proved difficult due to the lack of logical insights available in uncovering the best prompt.

Figure 5.17 illustrates the nuances associated with the prompt engineering process. Identical prompts, with the exception of an additional newline, yield completely different results.

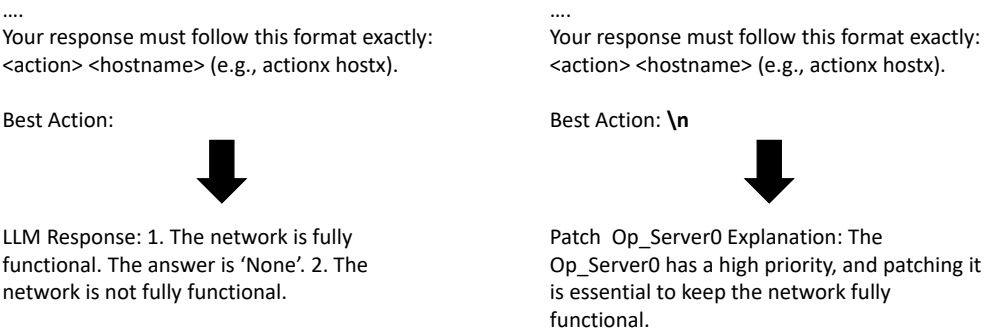


Figure 5.17: Illustration of the nuances with prompt engineering. The left shows an LLM response that cannot be reliably converted into an executable action. The right shows that adding a single newline rectifies this, producing an extractable action.

It was discovered that Cyber-Risk-Llama8B's existing training was inherently biased towards certain actions and hosts based on their names, despite other explicit instructions included in the prompt. For example, it would recommend the remove action over restore even though privileged processes were found on hosts, and prioritize enterprise servers over operational servers despite the explicit priority attribute associated with both. As such, the host names and actions were changed to generic names (e.g., action1, host1) which showed more stable performance.

It was observed that the LLM generally prioritized hosts that appeared earlier in the prompt. The likely cause for this is the fundamental masked self-attention mechanism used by decoder-only LLMs, where only prior tokens are used to calculate the attention score. This means that earlier tokens will have a greater impact, as they will be used in the attention score calculation for all future tokens.

Furthermore, the LLM appeared to struggle with extracting meaningful information pertaining to host attributes that used timesteps as their metric, such as the last analyzed field. This is logical, as Cyber-Risk-Llama8b likely was not trained on reinforcement learning specific data, making it difficult to map these metrics into something usable.

As such, the priority, last analyzed, and timestep fields were omitted from the prompt, and the priority was represented by placing the hosts in a specific order (operational server at the beginning).

To further increase performance, the minimum number of hops each host

had from the operational server was added to the prompt - but only for hosts along the critical paths (i.e., the shortest attack paths an attacker could follow to propagate its presence to the operational server from the user subnet).

Finally, the prompt was further optimized by listing explicit step-by-step instructions for the LLM to follow when making a decision. These included constraints such as never selecting the remove or restore action on hosts that do not have any suspicious processes or files.

The performance of the five prompts described above are shown in Figure 5.18. It can be seen that the performance is significantly better when explicit step-by-step instructions and constraints are included in the prompt.

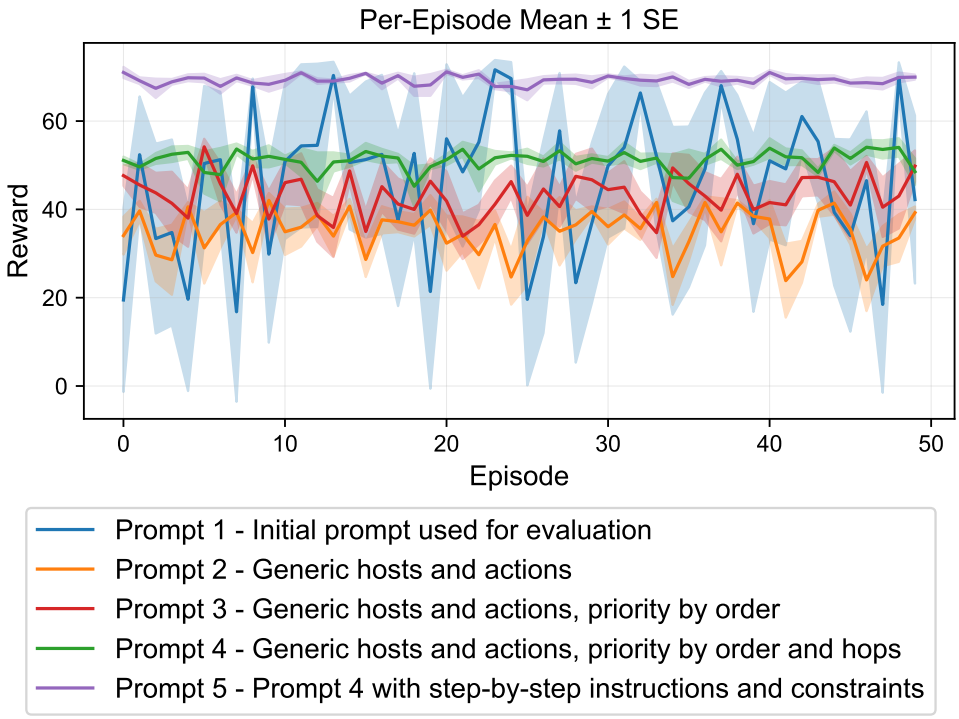


Figure 5.18: Evaluation of various prompts across 10 independent runs for 50 episodes. Per-episode mean reward with a ±1 standard error.

5.6.2 Standard Prompt Evaluation

Including explicit constraints and step-by-step instructions is not always feasible in complex environments. For this reason, the evaluation of the LLM integration was conducted using both the standard prompt (Prompt 4 in Figure 5.18)

and the optimized prompt (Prompt 5 in Figure 5.18). In addition to this, the standard prompt is used to provide a baseline that can be surpassed by independent RL, facilitating easier validation of the teacher-guided technique's success.

The gradually decaying, inference-only action masking coupled with a decaying auxiliary loss signal was identified as the best-performing teacher-guided strategy and was used for the LLM integration. For the standard prompt, a decay rate of 25% was applied to both the action mask and auxiliary loss every eight episodes (i.e., one training interval), with the decay beginning after four training intervals (32 episodes) of full teacher-guidance. The performance of the LLM-integrated agent compared to the PPO baseline is illustrated in Figure 5.19.

To encourage exploration beyond the teacher's policy, the entropy coefficient was increased by $5e^{-4}$ every time the auxiliary loss and action masking were decayed. Once the teacher's guidance was removed completely, the entropy coefficient was decreased by $2.5e^{-4}$ every training interval until it reached the baseline PPO agent's configuration.

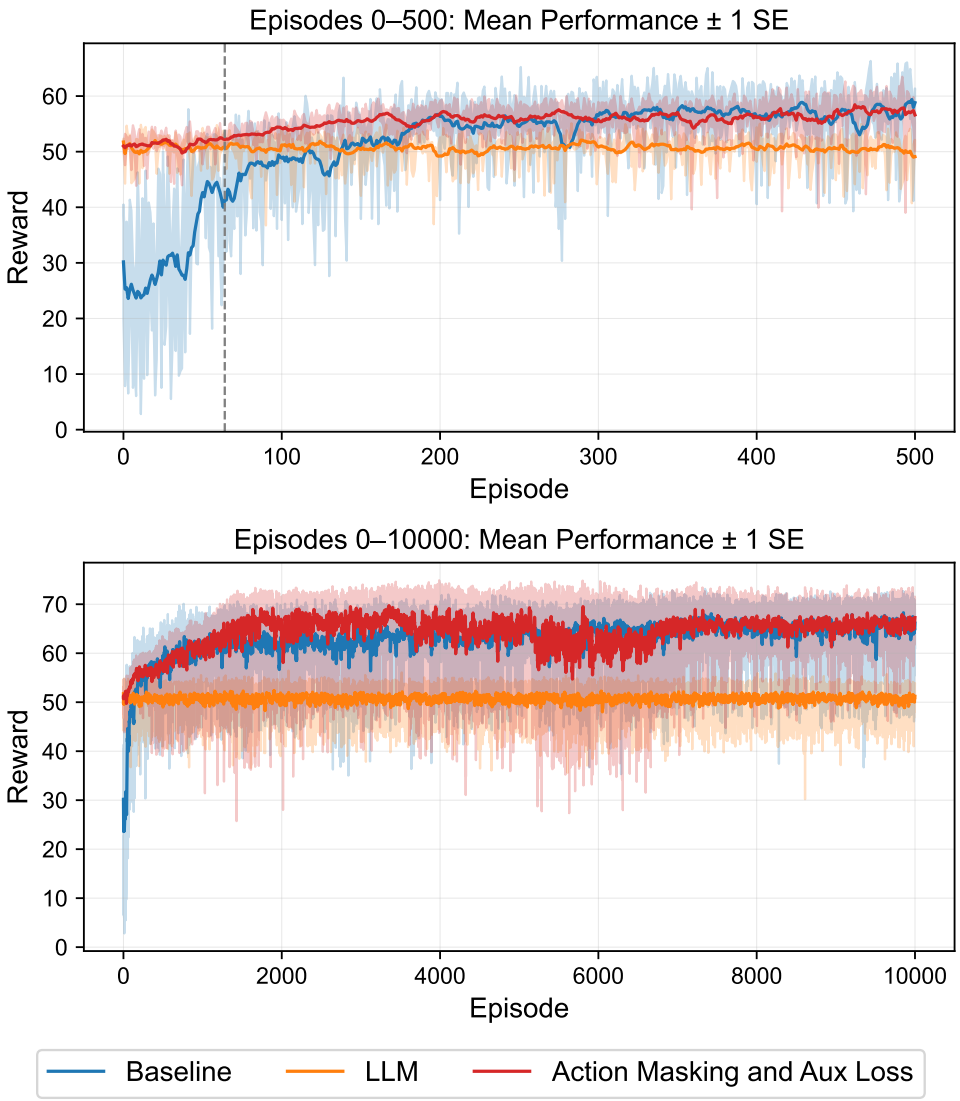


Figure 5.19: Comparison of LLM-guided training using action masking with auxiliary loss against the PPO baseline across 10 independent runs. The vertical dashed line indicates the point at which the agent has transitioned to fully independent RL (i.e., learning solely from the environment's signals). For clarity, the dashed line is only shown on the top plot.
Top: Mean reward after applying a 10-episode running average with a ±1 standard error for 500 episodes.
Bottom: Mean reward after applying a 10-episode running average with a ±1 standard error for 10,000 episodes.

Figure 5.19 shows high variance between runs for the baseline and the LLM-guided RL agent. This could be attributable to the stochasticity associated with the environment; however, there is noticeably more variation with the teacher-guided agent compared to the baseline. To validate if this is specific to the LLM, Figure 5.20 compares various metrics against leveraging a pretrained RL agent as the teacher, using an identical configuration.

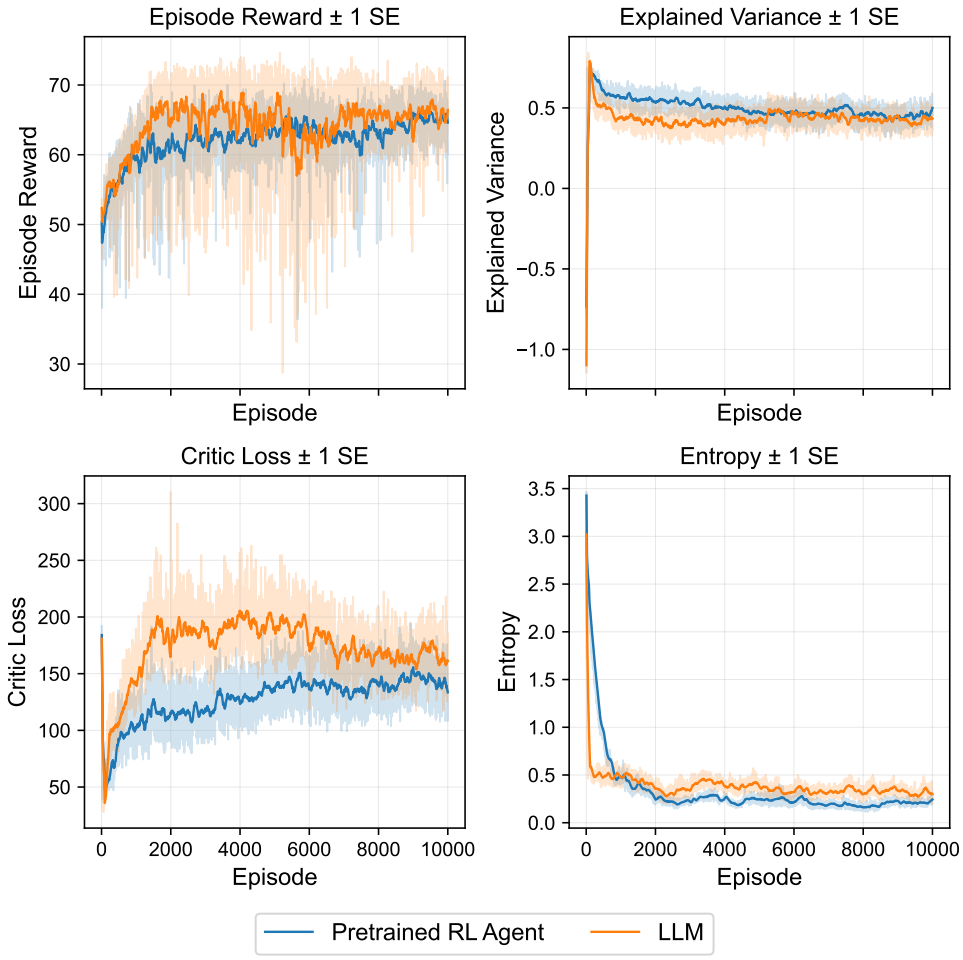


Figure 5.20: Comparison of LLM-guided training and pretrained RL agent guided training across 10 independent runs. Plots depict the mean for episode reward, explained variance, critic loss, and entropy across a 10-episode running average with a ±1 standard error.

While using the LLM as a teacher yields higher performance than the

pretrained RL agent (despite both teachers having similar performance individually), there is an apparent increase in instability across metrics, specifically episode reward, explained variance, and critic loss when using the LLM. A possible cause for this, is the fact that the LLM was never trained explicitly in the environment. In contrast, the pretrained RL agent was trained using identical manually engineered features and signals as the baseline RL agent, resulting in smoother updates to the critic's gradients.

This fundamental difference between the LLM's policy and a trained RL agent's policy is the reason that the decay was begun after four training intervals rather than a single interval (as was done in the pretrained agent evaluation). A noticeable drop in performance during the transition from teacher-guided to independent RL was observed if the decay began earlier.

5.6.3 Optimized Prompt Evaluation

The LLM integration was also evaluated using a stricter prompt that explicitly specifies what to do in a step-by-step format with constraints. For the optimized prompt, a decay rate of 10% was applied to both the action mask and the auxiliary loss signal every eight episodes (i.e., one training interval), with the decay beginning after thirty training intervals of full teacher-guidance. The entropy coefficient was increased by $2e^{-4}$ for every decay in auxiliary loss, and decreased by $1e^{-4}$ until it reached the baseline PPO agent's configuration once fully transitioned to independent RL. Figure 5.21 shows the performance of the optimized prompt against the PPO baseline.

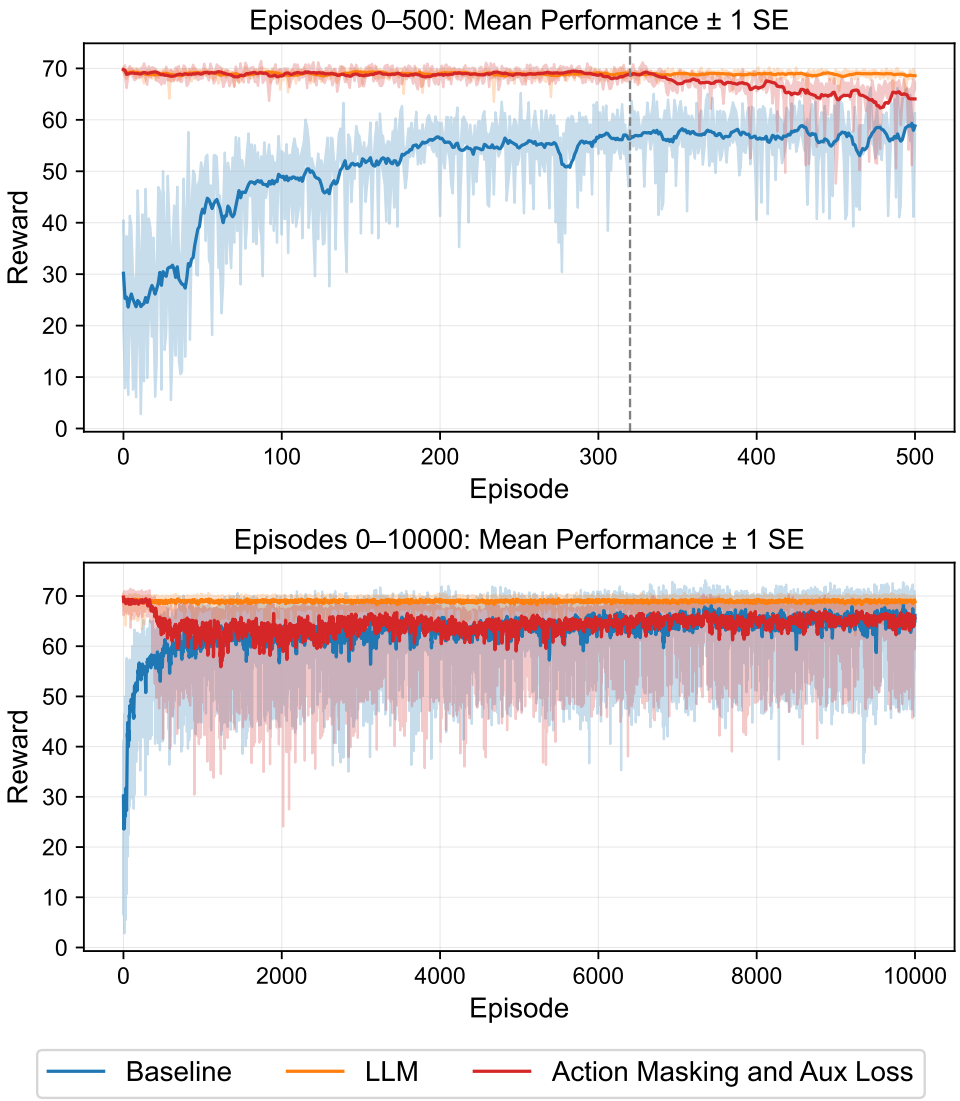


Figure 5.21: Comparison of LLM-guided training using action masking with auxiliary loss against the PPO baseline across 10 independent runs. The vertical dashed line indicates the point at which the agent has transitioned to fully independent RL (i.e., learning solely from the environment's signals). For clarity, the dashed line is only shown on the top plot.
Top: Mean reward after applying a 10-episode running average with a ±1 standard error for 500 episodes.
Bottom: Mean reward after applying a 10-episode running average with a ±1 standard error for 10,000 episodes.

The most notable observation in Figure 5.21 is the drop in performance as the RL agent is transitioned from LLM-guided training to independent RL. It can be seen that at around episode 1,200, the baseline reaches the LLM-guided agent's performance, and both exhibit similar behavior thereafter. It can also be seen that both RL agents do not converge to the LLM's baseline performance in 10,000 episodes (320,000 timesteps).

The drop in performance exhibited by the LLM-guided RL agent shown in Figure 5.21 can be caused by various factors, such as the actor not being able to map the manually engineered features of CybORG to the actions recommended by the LLM.

For validating the RL agent's ability to mimic the LLM's behavior, the same auxiliary loss technique was applied; however, training was halted completely after 30 training intervals (240 episodes). Furthermore, no action-masking was applied in this setup to confirm that the RL agent's policy is able to make these decisions without having its distributions directly modified. Figure 5.22 illustrates that the RL agent is able to achieve and maintain almost identical performance to the LLM, showing the capability of quickly distilling the 8-billion-parameter LLM's performance into a much cheaper 64,910-parameter feedforward neural network (0.0008114% the size).

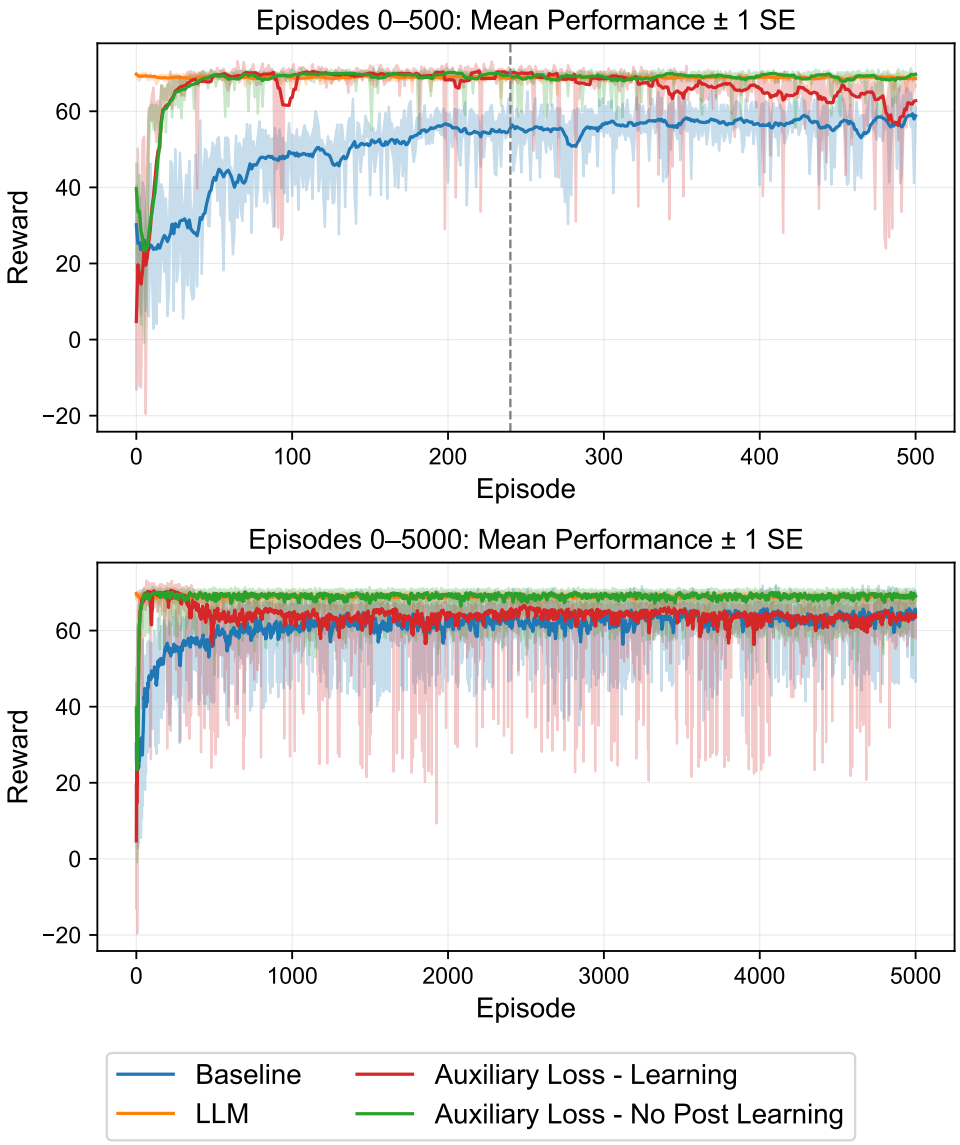


Figure 5.22: Comparing cutting off learning after the LLM-guided phase (distillation) and continuing to learn. The vertical dashed line indicates the point at which the LLM-guidance is abruptly removed for the distilled agent (the green curve), and where the LLM-guidance for the other agent (the red curve) begins to decay. For clarity, the dashed line is only shown on the top plot.
Top: Mean reward after applying a 10-episode running average with a ±1 standard error for 500 episodes.
Bottom: Mean reward after applying a 10-episode running average with a ±1 standard error for 5,000 episodes.

The drop in performance during the transition from LLM-guided to independent RL does not appear to be due to a lack of ability for the actor to map the manually engineered feature space to recommended actions, as shown by the successful distillation of knowledge into the RL agent. The other area to explore is the critic network. Because the critic network is trained almost exclusively on states that occurred as a result of executing LLM recommendations, it could have difficulty estimating the value of other actions.

The advantage, explained variance, critic loss, and entropy were recorded against the baseline agent's performance, as shown in Figure 5.23. Only the first 1,000 episodes are shown to focus on the transition from LLM-guided to independent learning.

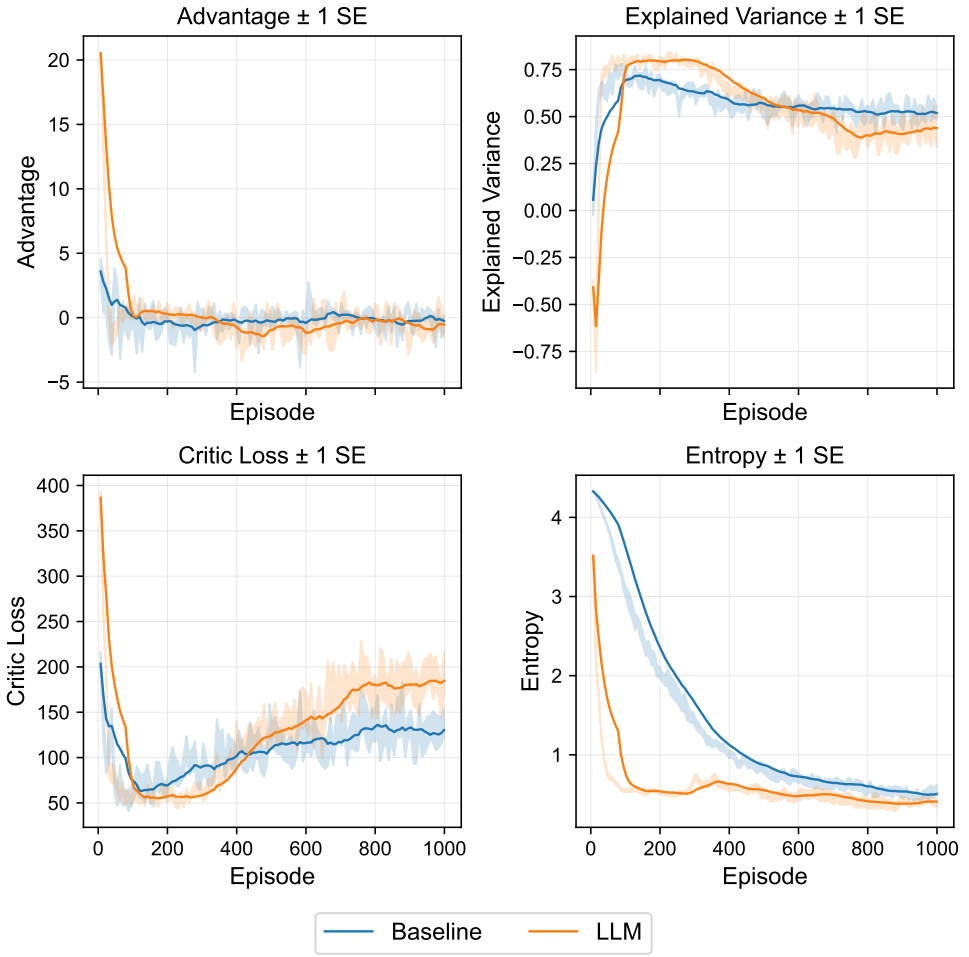


Figure 5.23: Comparison of LLM-guided training and independent RL across 10 runs for 1,000 episodes. Figures depict the mean for episode reward, explained variance, critic loss, and entropy across a 10-episode running average with a ±1 standard error.

These metrics show a noticeable shift during the transition from teacher-guided to independent RL, which starts at episode 240 and ends at episode 320. The advantage and explained variance decrease at this transition, while the critic loss increases, validating the hypothesis that the critic network struggles to make valid predictions for actions that do not perfectly align with the LLM's recommendations.

Entropy starts very low, demonstrating the deterministic nature of learning solely from the LLM, with a small but noticeable increase during the transition

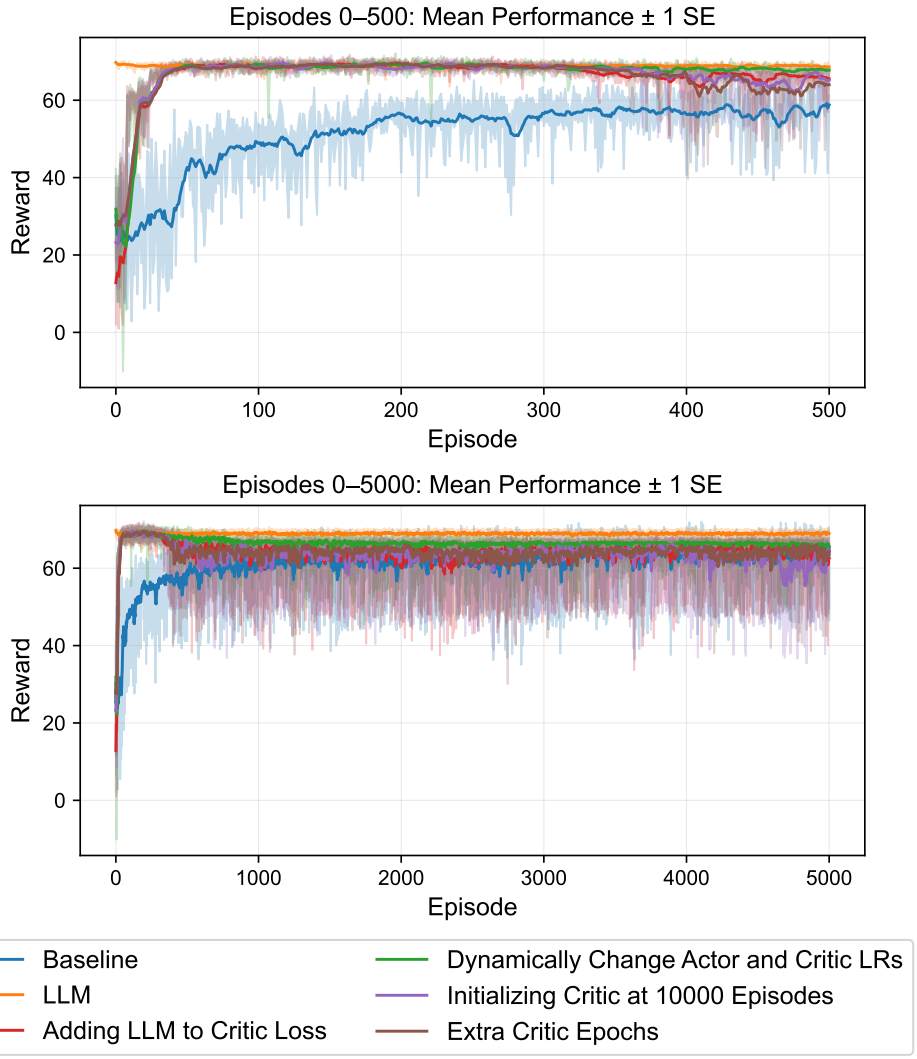
to independent learning. This increase during the transition can be attributed to the small entropy bonus that is added to encourage exploration in independent RL, but also due to the loss signals now incorporating advantages calculated using a struggling critic network.

In an attempt to rectify the apparent drop in performance during the transition from teacher-guided to independent RL, seven independent techniques were employed:

- Adding the LLM's recommendation to the critic loss.
- Replacing the critic network with a pretrained one.
- Dynamically changing the learning rates for the actor and critic.
- Adding extra epochs for the critic during the transition phase from teacher-guided to independent RL.
- Decaying the LLM's influence by a factor instead of a linear subtraction.
- Decaying the LLM's influence after 2,000 episodes instead of 320.
- Stopping the critic from learning after the transition to independent RL.

To keep the figures legible, these are split into two separate plots, where Figure 5.24a shows the first four techniques and Figure 5.24b shows the last three.

(a) Attempts 1-4 to Increase Stability



(b) Attempts 5-7 to Increase Stability

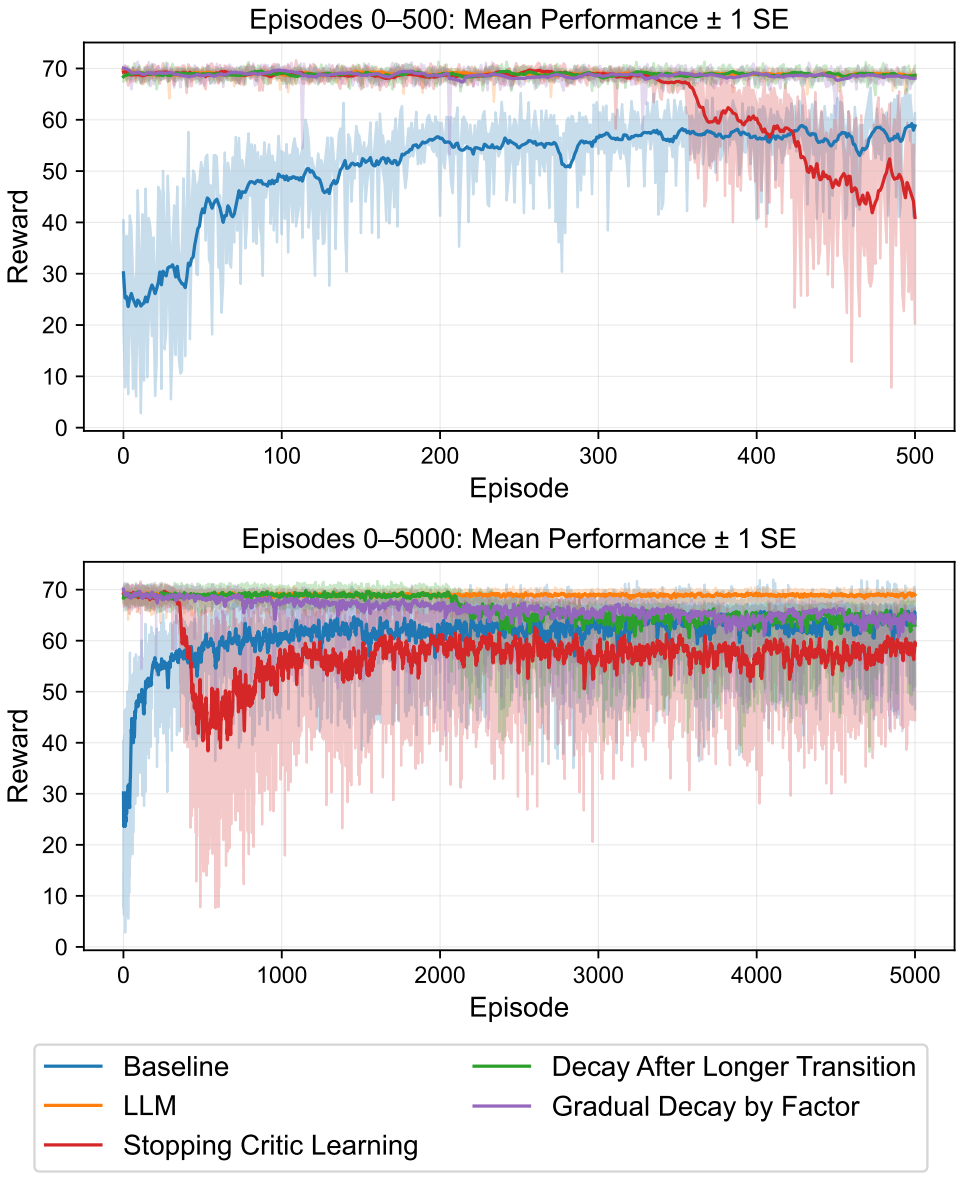


Figure 5.24: Attempting to facilitate a smoother transition from teacher-guided to independent RL using four different techniques shown in Figure 5.24a, and three different techniques shown in Figure 5.24b.
Figure 5.24a demonstrates: adding the LLM to the critic loss, initializing the critic with a pretrained model at episode 10,000, dynamically changing actor and critic LRs, and adding extra epochs for the critic network during the transition from teacher-guided to independent RL.
Figure 5.24b demonstrates: decaying the LLM influence after a longer transition, decaying the LLM's influence by a factor instead of a linear subtraction, and stopping the critic learning entirely after the transition.
Top graph for each plot: Mean reward after applying a 10-episode running average with a ± 1 standard error for 500 episodes.
Bottom graph for each plot: Mean reward after applying a 10-episode running average with a ± 1 standard error for 5,000 episodes.

The purpose of adding the LLM's recommendation as an auxiliary loss signal for the critic is to encourage the critic to put more emphasis on the LLM's recommendations, potentially improving the critic's stability during the transition from LLM-guided to independent RL. A factor of 0.1 was used for LLM's contribution to the critic loss, effectively reducing its impact by 90%. As discussed in the methodology section, this is the same MSE loss that is typically used by PPO [45]; however, masking was applied so that only samples that align with the LLM's recommendations were applied to the gradient.

Similarly, twice as many epochs were used for the critic network, but applied only during the decaying transition from teacher-guided to independent RL. This is to improve the critic's ability to estimate values for any new, unseen states during the RL agent's transition to independent learning.

For validation, the RL agent was also initialized with a critic network that was trained for 10,000 episodes. Although unrealistic in practice, the purpose of this was to validate whether a "warmed-up" critic network could facilitate a smoother transition to independent RL.

The three techniques described above show no noticeable improvement over the standard LLM-guided setup, exhibiting the same drop in performance.

The critic learning was increased from $1.6e^{-3}$ to $3.2e^{-3}$ to quickly estimate the returns of LLM-recommended actions, and then decreased to $0.8e^{-3}$ during the transition to independent RL. The actor learning rate was also decreased from $1.6e^{-3}$ to $0.8e^{-3}$ during the transition. As shown in Figure 5.24, this approach showed a slower drop in performance than the other three methods; however, this is due to the smaller actor learning rate and does not solve the fundamental problem, which is the decrease in performance during the transition.

Similarly, employing a gradual multiplicative decay factor of 0.99 per training interval instead of a linear subtraction, and waiting until episode 2,000 instead of 320 to commence the decay also slowed the rate of performance decline, but did not stop the decrease.

Finally, the learning for the critic network was stopped entirely after the transition to independent RL at episode 320. The idea here was to see if a critic trained exclusively on LLM recommendations would favor them enough to produce advantages that would push the actor back to the LLM's baseline policy. This shows the worst performance of all the attempts. When new states are introduced during the transition, the frozen critic is unable to adjust its parameters, and fails to produce reliable estimates. Eventually, the actor is able to stabilize, despite the suboptimal advantages returned from the critic at episode ≈1,550.

These results show that the drop in performance from the LLM-guided to independent RL is not solely attributable to the critic network.

Finally, to validate if the PPO baseline configuration can surpass the optimized LLM-guided RL agent, the baseline PPO agent was trained for 50,000 episodes (1.6 million timesteps) as shown in Figure 5.25.

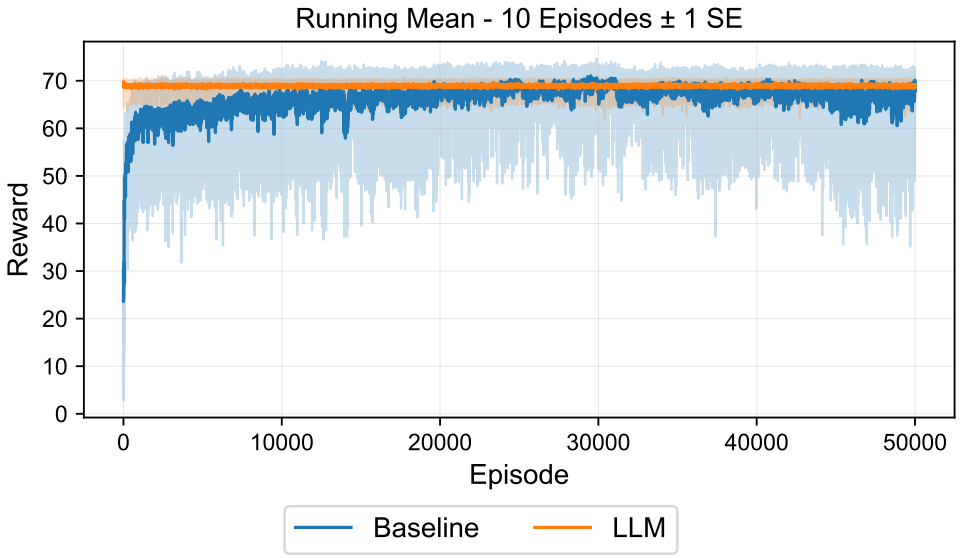


Figure 5.25: Comparing the optimized prompt against the baseline over 50,000 episodes across 10 independent runs using a 10-episode running average with a ±1 standard error. Due to the computational demands of an LLM, a distilled RL agent was used in its place.

From Figure 5.25, it can be seen that while individual runs of the baseline PPO agent surpass the LLM, its mean performance never appears to stabilize beyond the LLM. There are points after episode 22,000 where the mean of the baseline temporarily surpass the LLM; however, the performance does not remain stable.

5.6.4 Fundamental Limitation

Both of the implementations using the standard and optimized prompt show the same behavior: performance does not improve at the same rate as the baseline after the transition to independent RL. As a result, the baseline performance converges with the teacher-guided performance by approximately episode 180 for the standard prompt, and episode 1,200 for the optimized prompt. The fundamental issue appears to be the policy in which the agent converges on during the teacher-guided phase.

The difference in policies between an LLM-distilled RL agent with the standard and optimized prompt, compared to a baseline RL agent trained over 100 and 10,000 episodes is shown in Table 5.16.

Table 5.16: Comparing the top three probabilities between the distilled RL agents and ones trained until 100 and 10,000 episodes. Column titles are abbreviated for readability.

	Optimized	**Standard**	**100 Eps RL**	**10,000 Eps RL**
Largest Prob	99.56%	99.96%	21.64%	54.86%
Second Largest Prob	0.30%	0.03%	8.53%	21.91%
Third Largest Prob	0.08%	0.01%	5.52%	8.86%

Table 5.16 shows that the largest probability is approximately 33,000% bigger than the second largest for the RL agents trained using the LLM, whereas the difference in the top two probabilities for the trained RL agents is approximately 250%.

The auxiliary loss method employed has the agent modify its policy to prioritize a single action, resulting in a very peaked distribution, which does not occur naturally from independent RL. When the agent begins updating its policy using standard PPO loss, where a non-zero advantage is used, the policy will inevitably shift. Minor deviations to the logits associated with the highest ranked action result in significant changes to the lower ones to ensure a valid softmax is preserved.

Additionally, the critic was trained almost exclusively on states and transitions induced by the LLM's recommendations during the teacher-guided

phase. The initial limited exposure to the critic results in inaccurate, unreliable value predictions when new states are introduced during the switch from teacher-guided to independent RL.

A way to overcome the primary limitation is to have the LLM output a distribution of actions rather than a single recommendation. This would ensure that the policy is appropriately distributed during the teacher-guided phase, facilitating a smoother transition to independent PPO. A distribution can be obtained by querying the LLM individually for every possible action, as was done by Z. Zhou et al. [88]. This was deemed not realistic during the literature review due to an LLM's computational requirements and the environment's large action space, so the approach of mapping the LLM's tokens into a distribution was explored [88].

This study's process of mapping the LLM's output into a probability distribution and using it to guide training is detailed in Appendix E.

5.6.5 Discussion

As stated in Chapter 1, three criteria are used to evaluate the success of the LLM-guided agent compared to the baseline:

- Can it converge to an optimal policy quicker?
- Is its policy as accurate or more accurate?
- Does it initially produce more favorable actions?

When using the optimized prompt and providing the LLM explicit instructions and constraints for the environment, the RL agent is able to reach the same policy quicker. However, this accelerated convergence results from a performance drop during the transition from mimicking the LLM's behavior to learning solely from CybORG's reward signals. Integrating the LLM with the standard prompt, where its performance is considerably worse than the pretrained RL agent showed the same training time required to converge onto a favorable policy.

For the accuracy of the LLM-guided agent's policy using the standard and optimized prompt, there was no noticeable difference, except for the increased variance for the standard prompt.

The initial performance of the LLM-guided agents using both the standard and optimized prompts was notably better, enabling the agents to learn without having to explicitly perform obviously unfavorable actions.

The convergence speed and policy stability both show noticeable improvements when the LLM outputs a distribution instead of a single action. The details of this are discussed in Appendix E.

6 Conclusion

Autonomous Cyber Operations (ACO) is an innovative field that enables the training of agents to execute actions autonomously with minimal human oversight. Current ACO applications require these agents to learn from scratch, resulting in the need to perform obviously unfavorable actions to learn their associated consequences. The purpose of this study was to integrate external knowledge in the form of a Large Language Model (LLM) - that the agent could directly leverage - to eliminate the need to perform unfavorable actions and to increase training efficiency in terms of timesteps.

6.1 Contributions

The main contribution of this study is the demonstration of the positive impact that LLMs can have on training efficiency using CybORG's Cage Challenge 2 scenario. It showed that if prompted appropriately, the LLM can ultimately be used to increase training efficiency for an RL agent, with some caveats.

In addition to illustrating the positive impact LLMs can have on ACO, this study provides:

- A framework that can be used to efficiently evaluate and select the best-performing LLM for a particular environment.
- Modifications to CybORG's Cage Challenge 2 environment to better represent realistic cybersecurity conditions.
- A web application that can be used to visualize the process of training agents in the CybORG environment.
- A teacher-guided technique that combines action masking with auxiliary loss.
- The evaluation of various teacher-guided RL techniques in terms of how well they perform in a cybersecurity environment.
- A pipeline that can be leveraged to incorporate an LLM into the decision-making process for RL agents.

- The design of robust prompts that can enable LLMs to make effective and contextually relevant decisions in the context of CybORG.
- A method that can be employed to efficiently distill an LLM's generalized training into a much smaller RL agent.

6.2 Limitations

While this study has made contributions in the field of ACO, there are many limitations that exist throughout the various phases.

6.2.1 Selecting an LLM

While an autonomous, objective process was attempted, the evaluation for selecting an appropriate LLM ended up involving manually validating the responses of four LLMs for 100 questions against answers that were manually created. These answers were created without any formal criteria for what constitutes a best action, other than an estimation on how it would perform in CybORG. Furthermore, the answers were assigned a score of 1 for being aligned with the label, a 0.5 for relevant in the scenario and 0 otherwise. Instead, the quality of the answers outputted by an LLM should be critiqued on a continuous scale.

While the instantaneous reward from CybORG is not indicative of the best action, creating a process for comparing possible actions and evaluating them based on a spectrum of favorability would have improved the quality of the LLM evaluation.

Furthermore, a generic prompt was used across all LLMs to ensure a fair evaluation; however, due to the vast and different training data these LLMs were exposed to, some could have performed completely differently with minor prompt modifications. More effort spent in optimizing the prompts for each of the LLMs prior to the evaluation could have contributed to the quality of the selection process.

6.2.2 Environment Limitations

Perhaps the biggest limitation of this study is the simplicity of the simulation used to justify the potential relevance of LLM integration for practical ACO. Like most RL environments, CybORG functions by having the agents perform actions in a sequential manner; however, this is not representative of realistic cybersecurity scenarios, where attackers and defenders will wait for the other to execute an action. This could partially be rectified by allowing each agent

to interact with the environment for x timesteps before the other's turn, where x is a random integer.

The data outputted from CybORG is very simplistic and contains only new processes, files, or connections that are not in the host's baseline image, making it very easy to detect red activity. There is a green agent that can modify these; however, it also can only perform a single action every timestep, greatly under representing all the benign activity that would be present in an emulated network.

Because of the environment's straightforward observation space, the feature space engineering process for the RL agent is also quite simple and deterministic. For example, a host is considered exploited if the number of connections for a single port exceeds three.

While some stochasticity exists in the red agent that attacks the simulated network, it still follows the same pattern of scan, exploit, and privilege escalate for each of its hosts. In reality, a wide variety of attack paths exist, such as phishing, that are not present in this scenario. CybORG's multi-agent RL environment (Cage Challenge 4 [52]) helps alleviate some of these concerns, where a green agent can become red, representing compromises through social engineering.

While this study provides a strong proof of concept for the benefits of incorporating an LLM to augment decision-making in ACO, the extent of its impact in real-world settings cannot be guaranteed due to the inherent limitations of using a simulated RL environment for cybersecurity.

It should be noted that perfectly simulating the complex, dynamic nature of cybersecurity, while providing signals for RL agents to learn is a very difficult - perhaps impossible - challenge. Overall, the CybORG environment is very impressive and considered one of the best simulated cybersecurity environments for ACO at the time of writing [2]; however, it is still lacking substantial information that would be present in a real network.

6.2.3 Agent Evaluation Limitations

The evaluation of the Proximal Policy Optimization (PPO) and Deep Q-Network (DQN) agents was sufficient for selecting the best baseline, but could have been more exhaustive. While many combinations of parameters were tested for each algorithm, the tests could have been run for longer before concluding that PPO is more effective in CybORG. While 150 episodes provided a useful indication of convergence, running these tests for longer may have shown different outcomes - for example, allowing the DQN agent to transition to solely sampling from its Q-table (when $\epsilon \approx 0$).

6.2.4 Teacher-Guided Integration Limitations

More hyperparameter tuning could have been performed for evaluating the various teacher-guided techniques. Unlike the baseline agent development, which included testing many combinations in a structured fashion, hyperparameter tuning for this phase was conducted in an ad-hoc manner.

Furthermore, while incorporating a pretrained RL agent instead of an LLM helped focus on the technique itself (and reduced time due to the lower computational demand), the LLM could have been evaluated for each of these techniques in addition to the RL agent.

6.2.5 Parsing CybORG's Output

One of the arguments for the potential benefits of incorporating an LLM into the RL pipeline, was that it eliminated the need to manually engineer features from CybORG's raw state space, greatly reducing the risk of missing vital information.

However, "pseudo" feature engineering was conducted with CybORG's raw state space to parse it into a format in which patterns could easily be extracted by an LLM. The end format omitted information, such as host operating systems and architectures.

Although an apparent decrease in performance was observed when these were included in the prompt, more comprehensive prompt engineering that included these details could have been beneficial.

6.2.6 Final Prompt Design

The final prompt design used for the LLM involved replacing the actions and hosts with generic names, and defining each action as it relates to CybORG. The purpose of the LLM selection in Phase 1 was to evaluate the models' existing cybersecurity knowledge and how this could positively impact RL training for CybORG. Defining the actions and the rules for CybORG within the prompt defeats the purpose of evaluating the LLMs' existing knowledge, focusing instead on their pattern extraction capabilities.

The evaluation process in Phase 1 should have included all of this information in the prompts, to test the LLMs' abilities to contextualize patterns in CybORG as opposed to solely validating whether their learned definitions for the actions and hosts map well to the environment.

6.2.7 LLM Resource Requirements

The primary purpose of this study was to integrate an LLM to increase the training efficiency of RL agents. It used the number of timesteps as the metric for evaluating this, completely ignoring the computational requirements to run an LLM, and the time required for it to generate a response. For example, for new states, the LLM takes an average of 3.685 seconds (as shown in Table 5.1) to produce a response, as opposed to the fraction of a second required by an RL agent (caching was enabled for the LLM, greatly increasing the speed for previously seen states). The selected LLM also has 8 billion parameters, each represented by a 16-bit float, meaning that 16 GiB of memory are required just to have it loaded. On the contrary, the pretrained RL agent needs 128 KiB. It should be noted that these figures are just for having the models loaded, not the GPU requirements to have them generate responses.

6.2.8 LLM Integration Limitations

As mentioned in Chapter 5, the single-action integration technique used for the LLM was fundamentally flawed, encouraging the RL agent to converge to a narrow policy that does not stabilize well in PPO, while fixating the critic network to learn the values of states produced by LLM recommendations, completely ignoring others. This ultimately resulted in a non-optimal transition from teacher-guided to independent RL, where the agent's policy - initially shaped by the LLM's single-action recommendations - had to be flattened for further refinement.

While the solution in Appendix E partially rectifies this problem by mapping the LLM's output into a probability distribution, it still results in similar critic instability issues, stemming from having it initially learn only on states induced by actions within the LLM's recommended probability distribution.

6.2.9 Insufficient Testing for Transferability

One of the arguments made in Chapter 1 and Chapter 3 is that the LLM can be easily transferred due to its vast amount of training, increasing training efficiency across different environments. Although the LLM integration was evaluated on nine different scenarios ranging from four to twelve hosts (ref Appendix F), these scenarios consisted of identical functionality with minor modifications to CybORG's default scenario and red agent. To prove the transferability of the LLM, different cybersecurity environments with a different action space and a different set of rules should have been used.

6.3 Future Work

Although there are many limitations with this study, the novel contributions it has made lay the groundwork for significant advances in ACO. These opportunities include:

- Modifying the integration technique and LLM's response to facilitate a smoother transition from teacher-guided to independent RL. This could also include optimizing the prompt engineering process to include more advanced techniques like Retrieval-Augmented Generation (RAG) [39].
- Integrating the LLM into the emerging multi-agent RL field [89].
- Leveraging the LLM to augment decision-making for the red agent.
- Fine-tuning the LLM to tailor its output to the CybORG environment.
- Integrating the LLM into the feature engineering process.
- Leveraging the LLM to augment decision-making in an emulated environment.
- Incorporating an encoder-only LLM into the decision-making process (with more success than what was attempted in Appendix D).
- Defending the LLM in an adversarial setting.

The integration of an LLM into the RL pipeline to augment decision-making can be a key milestone in ACO applications, creating a strong foundation for future advancements in autonomous cybersecurity.

Bibliography

[1] Nilotpal Chakraborty. INTRUSION DETECTION SYSTEM AND INTRUSION PREVENTION SYSTEM: A COMPARATIVE STUDY. *International Journal of Computing and Business Research*, 4(2), 2013.

[2] Callum Baillie, Maxwell Standen, Jonathon Schwartz, Michael Docking, David Bowman, and Junae Kim. CybORG: An Autonomous Cyber Operations Research Gym, February 2020.

[3] Mariah St. John. Cybersecurity stats: Facts and figures you should know. Forbes, 2024.

[4] Sabih Saeed, Paul Black, Shaoning Pang, and Peter Vamplew. A Review of Reinforcement Learning Enabled Autonomous Cyber Operations Platforms, 2024. Publisher: SSRN.

[5] Center for Security and Emerging Technology, Micah Musser, and Ashton Garriott. Machine Learning and Cybersecurity: Hype and Reality. Technical report, Center for Security and Emerging Technology, June 2021.

[6] Jacob Wiebe, Ranwa Al Mallah, and Li Li. Learning Cyber Defence Tactics from Scratch with Multi-Agent Reinforcement Learning, August 2023. arXiv:2310.05939 [cs].

[7] Garrett McDonald. *Competitive Reinforcement Learning for Autonomous Cyber Operations*. PhD thesis, Royal Military College of Canada, Kingston, Ontario, May 2023.

[8] CyberNative. Cyberbase 13b. `https://huggingface.co/CyberNative/CyberBase-13b`, 2024.

[9] mlgwad. Cyberdost 2b v2.0. `https://huggingface.co/mlgawd/navarasa-2b-2.0-cyberdost`, 2024.

[10] segolilylabs. Lily cybersecurity 7b v0.2. `https://huggingface.co/segolilylabs/Lily-Cybersecurity-7B-v0.2`, 2024.

[11] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A Comprehensive Overview of Large Language Models, April 2024. arXiv:2307.06435 [cs].

[12] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. BERTScore: Evaluating Text Generation with BERT. International Conference on Learning Representations, ICLR, February 2020. arXiv:1904.09675 [cs].

[13] Mark Pfeiffer, Samarth Shukla, Matteo Turchetta, Cesar Cadena, Andreas Krause, Roland Siegwart, and Juan Nieto. Reinforced Imitation: Sample Efficient Deep Reinforcement Learning for Mapless Navigation by Leveraging Prior Demonstrations. *IEEE Robotics and Automation Letters*, 3(4):4423–4430, October 2018.

[14] Ali Beikmohammadi and Sindri Magnusson. TA-Explore: Teacher-Assisted Exploration for Facilitating Fast Reinforcement Learning. London, United Kingdom, May 2023. 2023 International Foundation for Autonomous Agents and Multiagent Systems.

[15] Ziyi Wang, Xinran Li, Luoyang Sun, Haifeng Zhang, Hualin Liu, and Jun Wang. Learning State-Specific Action Masks for Reinforcement Learning. *Algorithms*, 17(2):60, February 2024. Number: 2 Publisher: Multidisciplinary Digital Publishing Institute.

[16] Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. Reward Design with Language Models. International Conference on Learning Representations, ICLR, February 2023. arXiv:2303.00001 [cs].

[17] Songjun Tu, Jingbo Sun, Qichao Zhang, Xiangyuan Lan, and Dongbin Zhao. Online Preference-based Reinforcement Learning with Self-augmented Feedback from Large Language Model, December 2024. arXiv:2412.16878 [cs] version: 1.

[18] Liangliang Chen, Yutian Lei, Shiyu Jin, Ying Zhang, and Liangjun Zhang. RLingua: Improving Reinforcement Learning Sample Efficiency in Robotic Manipulations With Large Language Models. *IEEE Robotics and Automation Letters*, 9(7):6075–6082, July 2024. Conference Name: IEEE Robotics and Automation Letters.

[19] Zihao Zhou, Hu Bin, Zhao Chenyang, Lu Bin, and Zhang Pu. Large Language Model as a Policy Teacher for Training Reinforcement Learning Agents. In *IJCAI International Joint Conference on Artificial Intelligence*, pages 5671–5679, Jeju, Korea, 2024. International Joint Conferences on Artificial Intelligence.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[21] Robin M. Schmidt. Recurrent Neural Networks (RNNs): A gentle Introduction and Overview, November 2019. arXiv:1912.05911 [cs, stat].

[22] Qing Luo, Wei Zeng, Manni Chen, Gang Peng, Xiaofeng Yuan, and Qiang Yin. Self-Attention and Transformers: Driving the Evolution of Large Language Models. In *2023 IEEE 6th International Conference on Electronic Information and Communication Technology (ICEICT)*, pages 401–405. Institute of Electrical and Electronics Engineers Inc., July 2023. ISSN: 2836-7782.

[23] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[24] Jesse Roberts. How Powerful are Decoder-Only Transformer Neural Models? In *2024 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, June 2024. arXiv:2305.17026 [cs].

[25] Gokul Yenduri, Ramalingam M, Chemmalar Selvi G, Supriya Y, Gautam Srivastava, Praveen Kumar Reddy Maddikunta, Deepti Raj G, Rutvij H. Jhaveri, Prabadevi B, Weizheng Wang, Athanasios V. Vasilakos, and Thippa Reddy Gadekallu. Generative Pre-trained Transformer: A Comprehensive Review on Enabling Technologies, Potential Applications, Emerging Challenges, and Future Directions, May 2023. arXiv:2305.10435 [cs].

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, May 2019. arXiv:1810.04805 [cs].

[27] Gemini Team. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, December 2024. arXiv:2403.05530 [cs].

[28] OpenAI. GPT-4 Technical Report, March 2024. arXiv:2303.08774 [cs].

[29] Perrault Ray and Clark Jack. Artificial intelligence index report 2024. Technical report, Standford University.

[30] Ming Shen. Rethinking Data Selection for Supervised Fine-Tuning, February 2024. arXiv:2402.06094 [cs].

[31] Nisan Stiennon, Long Ouyang, Jeff Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul Christiano. Learning to summarize from human feedback. Neural information processing systems foundation, February 2022. arXiv:2009.01325 [cs].

[32] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct Preference Optimization: Your Language Model is Secretly a Reward Model, July 2024. arXiv:2305.18290 [cs].

[33] Parthasarathy Balavadhani, Zafar Ahtsham, Khan Aafaq, and Shahid Arsalan. The Ultimate Guide to Fine-Tuning LLMs from Basics to Breakthroughs: An Exhaustive Review of Technologies, Research, Best Practices, Applied Research Challenges and Opportunities, 2024.

[34] Sidji Matthew and Stephenson Matthew. Prompt Engineering ChatGPT for Codenames. IEEE Computer Society, September 2024.

[35] Noor Nashid, Mifta Sintaha, and Ali Mesbah. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2450–2462, Melbourne, Australia, May 2023. IEEE Computer Society.

[36] Gregory Palmer, Chris Parry, Daniel J. B. Harrold, and Chris Willis. Deep Reinforcement Learning for Autonomous Cyber Operations: A Survey, September 2024. arXiv:2310.07745 [cs].

[37] Eman Jawad. THE DEEP NEURAL NETWORK-A REVIEW. *IJRDO -JOURNAL OF MATHEMATICS*, 9:1–5, September 2023.

[38] OpenAI. Chatgpt (may 14 version). `https://openai.com/chatgpt`, 2025. Accessed: May 20, 2025.

[39] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-Augmented Generation for Large Language Models: A Survey, March 2024. arXiv:2312.10997 [cs].

[40] Martin Heller. Reinforcement learning explained, June 2019. Publisher: IDG Communications, Inc.

[41] Wang Qiang and Zhan Zhongli. Reinforcement learning model, algorithms and its application. In *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*, pages 1143–1146. IEEE Computer Society, August 2011.

[42] Qingyan Huang. Model-Based or Model-Free, a Review of Approaches in Reinforcement Learning. In *2020 International Conference on Computing and Data Science (CDS)*, pages 219–221. Institute of Electrical and Electronics Engineers Inc., August 2020.

[43] Jingkai Jia and Wenlin Wang. Review of reinforcement learning research. In *2020 35th Youth Academic Annual Conference of Chinese Association of Automation (YAC)*, pages 186–191. IEE Computer Society, October 2020.

[44] Le Lyu, Yang Shen, and Sicheng Zhang. The Advance of Reinforcement Learning and Deep Reinforcement Learning. In *2022 IEEE International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA)*, pages 644–648. Institute of Electrical and Electronics Engineers Inc., February 2022.

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, August 2017. arXiv:1707.06347 [cs].

[46] Li Li, Jean-Pierre S El Rami, Ryan Kerr, Adrian Taylor, and Grant Vandenberghe. Towards Autonomous Cyber Operation Agents: Exploring the Red Case, September 2023.

[47] Mitchell Kiely, David Bowman, Maxwell Standen, and Christopher Moir. On autonomous agents in a cyber defence environment. *arXiv preprint arXiv:2309.07388*, 2023.

[48] Defence Science and Technology Group. The technical cooperation program. `https://www.dst.defence.gov.au/partnership/technical-cooperation-program`, n.d.

[49] Mitchell Kiely, David Bowman, Maxwell Standen, and Christopher Moir. Cage challenge 1. https://github.com/cage-challenge/cage-challenge-1, n.d.

[50] Mitchell Kiely, David Bowman, Maxwell Standen, and Christopher Moir. Cage challenge 2. https://github.com/cage-challenge/cage-challenge-2, n.d.

[51] Mitchell Kiely, David Bowman, Maxwell Standen, and Christopher Moir. Cage challenge 3. https://github.com/cage-challenge/cage-challenge-3, n.d.

[52] Mitchell Kiely, David Bowman, Maxwell Standen, and Christopher Moir. Cage challenge 4. `https://github.com/cage-challenge/cage-challenge-4`, n.d.

[53] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents. volume 162 of *Proceedings of Machine Learning Research*, pages 9118–9147. ML Research Press, March 2022.

[54] Naman Goyal Jingfei Du Mandar Joshi Danqi Chen Omer Levy Mike Lewis Luke Zettlemoyer Yinhan Liu, Myle Ott and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv, 2019.

[55] 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 Tom B. Brown, Benjamin Mann and Dario Amodei. Language models are few-shot learners. arXiv preprint arXiv, 2020.

[56] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods, October 2018. arXiv:1802.09477 [cs].

[57] Bowen Wang Chenhui Zhang Da Yin Dan Zhang Diego ROJAS Guanyu Feng Hanlin Zhao Hanyu Lai Hao Yu Hongning Wang Jiadai Sun Jiajie Zhang Jiale Cheng Jiayi Gui Jie Tang Jing Zhang Jingyu Sun Juanzi Li Lei Zhao Lindong Wu Lucen Zhong Mingdao Liu Minlie Huang Peng Zhang Qinkai Zheng Rui Lu Shuaiqi Duan Shudan Zhang Shulin Cao Shuxun Yang Weng Lam Tam Wenyi Zhao Xiao Liu Xiao Xia Xiaohan Zhang Xiaotao Gu Xin Lv Xinghan Liu Xinyi Liu Xinyue Yang Xixuan Song Xunkai Zhang Yifan An Yifan Xu Yilin Niu Yuantao Yang Yueyan Li Yushi Bai Yuxiao Dong Zehan Qi Zhaoyu Wang Zhen Yang Zhengxiao Du Zhenyu Hou Aohan Zeng, Bin Xu and Zihan Wang. Chatglm: A family of large language models from glm-130b to glm-4 all tools. Zhipu AI, July 2024.

[58] Zi Lin Wei-Lin Chiang, Zhuohan Li and et. al. Vicuna: An open-source chatbot impressing gpt-4 with 90

[59] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision Transformer: Reinforcement Learning via Sequence Modeling. volume 18 of *Advances in Neural Information Processing Systems*, pages

15084–15097. Neural information processing systems foundation, June 2021.

[60] Jiawei Wang, Teng Wang, Wenzhe Cai, Lele Xu, and Changyin Sun. Boosting Efficient Reinforcement Learning for Vision-and-Language Navigation With Open-Sourced LLM. *IEEE Robotics and Automation Letters*, 10(1):612–619, January 2025. Conference Name: IEEE Robotics and Automation Letters.

[61] Michael Guastalla, Yiyi Li, Arvin Hekmati, and Bhaskar Krishnamachari. Application of Large Language Models to DDoS Attack Detection. In Yu Chen, Chung-Wei Lin, Bo Chen, and Qi Zhu, editors, *Security and Privacy in Cyber-Physical Systems and Smart Vehicles*, volume 552, pages 83–99. Springer Nature Switzerland, 2024. Series Title: Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering.

[62] Tarek Ali and Panos Kostakos. HuntGPT: Integrating Machine Learning-Based Anomaly Detection and Explainable AI with Large Language Models (LLMs), September 2023. arXiv:2309.16021 [cs].

[63] Johannes F. Loevenich, Erik Adler, Rémi Mercier, Alexander Velazquez, and Roberto Rigolin F. Lopes. Design of an Autonomous Cyber Defence Agent using Hybrid AI models. In *2024 International Conference on Military Communication and Information Systems (ICMCIS)*, pages 1–10, April 2024.

[64] A Bhagyalakshmi, C D Sruthi Laya, A M Yoga Preethikaa, and V Varsha. Machine Learning based Early Detection of Ongoing Cyber-Attacks. In *2024 3rd International Conference on Applied Artificial Intelligence and Computing (ICAAIC)*, pages 766–771. Institute of Electrical and Electronics Engineers Inc., June 2024.

[65] Muhammad Omer Farooq and Thomas Kunz. A Generic Blue Agent Training Framework for Autonomous Cyber Operations. In *2024 IFIP Networking Conference (IFIP Networking)*, pages 515–521. Institute of Electrical and Electronics Engineers Inc., June 2024. ISSN: 1861-2288.

[66] Ben Goertzel, Cassio Pennachin, Dov M. Gabbay, Jörg Siekmann, A. Bundy, J. G. Carbonell, M. Pinkal, H. Uszkoreit, M. Veloso, W. Wahlster, and M. J. Wooldridge, editors. *Artificial General Intelligence*. Cognitive Technologies. Springer, Berlin, Heidelberg, 2007.

[67] Akram Awad, Littig Lars, and Sophie Geraerdts. Artificial General Intelligence - White Paper. Technical report, September 2024.

[68] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of Artificial General Intelligence: Early experiments with GPT-4, April 2023. arXiv:2303.12712 [cs].

[69] Pat Langley, John E. Laird, and Seth Rogers. Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*, 10(2):141–160, June 2009.

[70] Soumil Rathi. Approaches to Artificial General Intelligence: An Analysis. Publisher: arXiv.

[71] Jie Zhang, Hui Wen, Liting Deng, Mingfeng Xin, Zhi Li, Lun Li, Hongsong Zhu, and Limin Sun. HackMentor: Fine-Tuning Large Language Models for Cybersecurity. In *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 452–461, Exeter, United Kingdom, November 2023. IEEE.

[72] Vanessasml. `https://huggingface.co/Vanessasml/cyber-risk-llama-3-8b`, 2024.

[73] Venky. `https://huggingface.co/ZySec-AI/SecurityLLM`, 2024.

[74] Sabit Ekin. Prompt Engineering For ChatGPT: A Quick Guide To Techniques, Tips, And Best Practices, May 2023.

[75] Hakan T. Otal and M. Abdullah Canbaz. LLM Honeypot: Leveraging Large Language Models as Advanced Interactive Honeypot Systems. In *2024 IEEE Conference on Communications and Network Security (CNS)*, pages 1–6, September 2024. ISSN: 2994-5895.

[76] Suran Abhishek. Reinforcement_learning/Atari_dqn_image.ipynb at master · abhisheksuran/Reinforcement_learning · GitHub.

[77] Keon. deep-q-learning/dqn_batch.py at master · keon/deep-q-learning.

[78] Antonin RAFFIN, Quentin Gallouédec, Noah Dormann, Adam Gleave, Anssi, Alex Pasquali, Juan Rocamonde, M. Ernestus, Patrick Helm, Corentin, Quinn Sinclair, Thomas Simonini, Tobias Rohrer, Sidney Tio, Rohan Tangri, Tom Dörr, Wilson, Steven H. Wang, Sam Toyer, Roland Gavrilescu, Paul Maria Scheikl, Parth Kothari, Oleksii Kachaiev, Bernhard Raml, Chris Schindlbeck, Costa Huang, Dominic Kerr, Grégoire Passault, Jan-Hendrik Ewers, and Marc Duclusaud. DLR-RM/stable-baselines3: v2.5.0: New algorithm (SimBa in SBX) and NumPy 2.0 support, January 2025.

[79] Phil Tabor. Youtube-Code-Repository/ReinforcementLearning/PolicyGradient/PPO/torch at master · philtabor/Youtube-Code-Repository, 2021.

[80] Adrian Holovaty and Simon Willson. Django, 2024.

[81] sqlite3 — DB-API 2.0 interface for SQLite databases.

[82] Nick Downie. Chart.js, 2024.

[83] Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. ACTIVATION FUNCTIONS IN NEURAL NETWORKS. *International Journal of Engineering Applied Sciences and Technology*, 04(12):310–316, May 2020.

[84] Optuna - A hyperparameter optimization framework.

[85] Douglas G Altman and J Martin Bland. Standard deviations and standard errors. *BMJ : British Medical Journal*, 331(7521):903, October 2005.

[86] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Why Should I Trust You?": Explaining the Predictions of Any Classifier, August 2016. arXiv:1602.04938 [cs] version: 3.

[87] Jonathan Hui. RL — Tips on Reinforcement Learning, February 2023.

[88] Jiehan Zhou, Yang Zhao, Jiahong Liu, Peijun Dong, Xiaoyu Luo, Hang Tao, Shi Chang, and Hanjiang Luo. LLM4RL: Enhancing Reinforcement Learning with Large Language Models. In *2024 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 86–88. Institute of Electrical and Electronics Engineers Inc., August 2024. ISSN: 2576-7046.

[89] Contractor Faizan. *Learning to Communicate in Multi-Agent Reinforcement Learning for Autonomous Cyber Defence*. PhD thesis, Royal Military College of Canada, Kingston, Ontario, May 2024.

[90] Aghaei Ehsan. ehsanaghaei/SecureBERT · Hugging Face.

[91] Jiequan Cui, Beier Zhu, Qingshan Xu, Zhuotao Tian, Xiaojuan Qi, Bei Yu, Hanwang Zhang, and Richang Hong. Generalized Kullback-Leibler Divergence Loss, March 2025. arXiv:2503.08038 [cs].

Appendices

A CybORG Sequence Diagrams

Chapter 2 discussed the agents' interaction with the environment at a high-level; however, there are many underlying complexities that were not mentioned. Figures A.1 and A.2 illustrate all entities and processes involved for initializing the environment and executing an action. While an in-depth knowledge of this process is not essential to understand this study, it is important to highlight the effort the Technical Cooperation Program (TCP) put into creating a simulated environment that mimics cybersecurity.

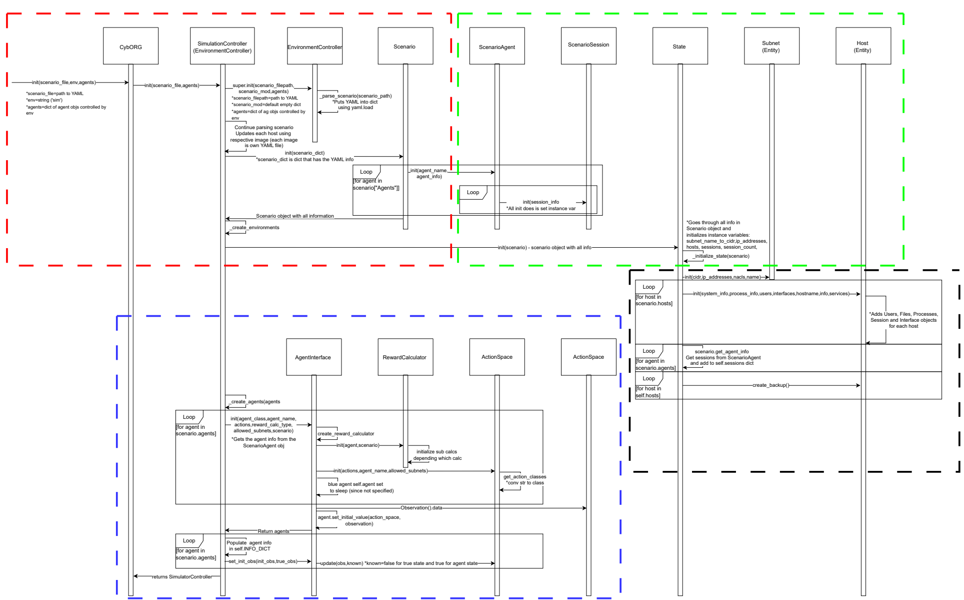
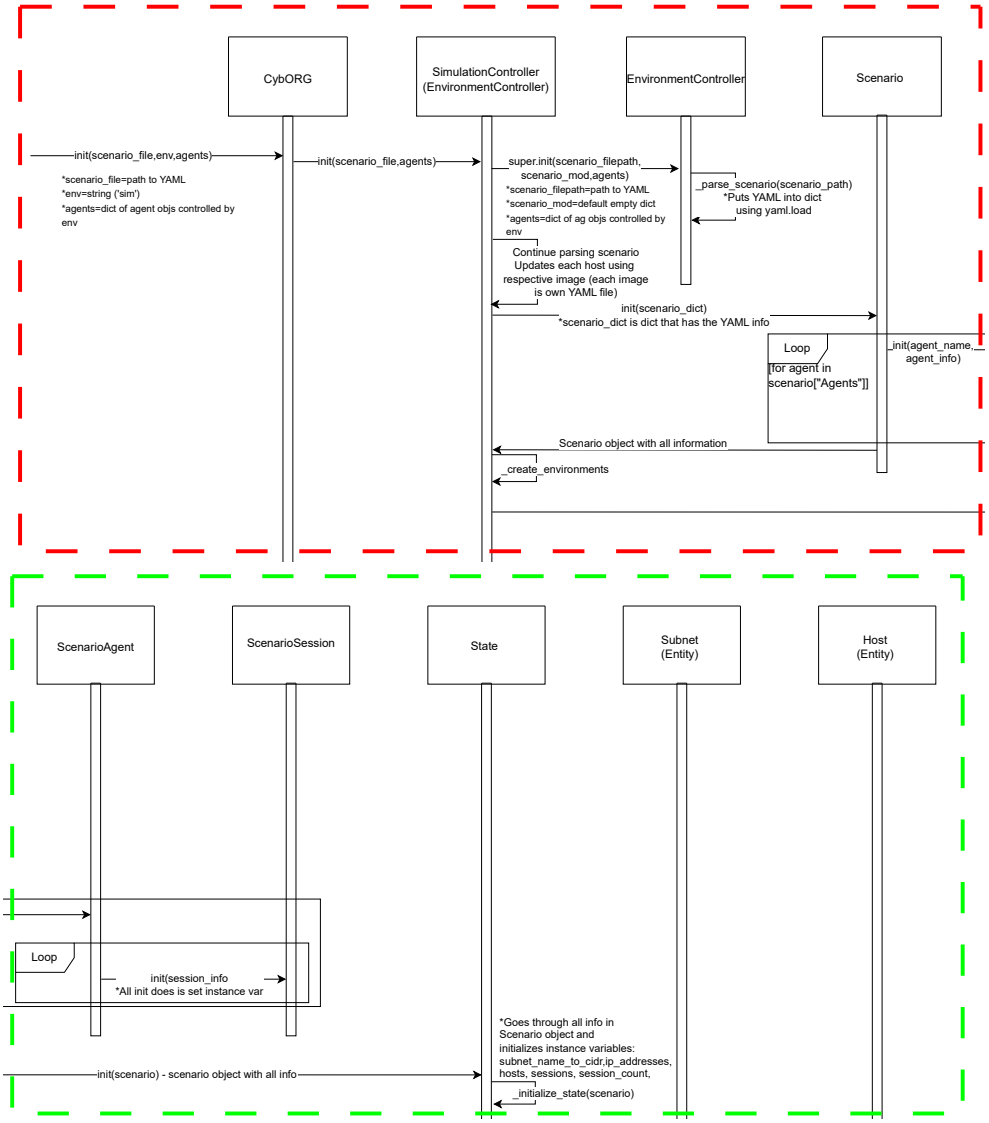
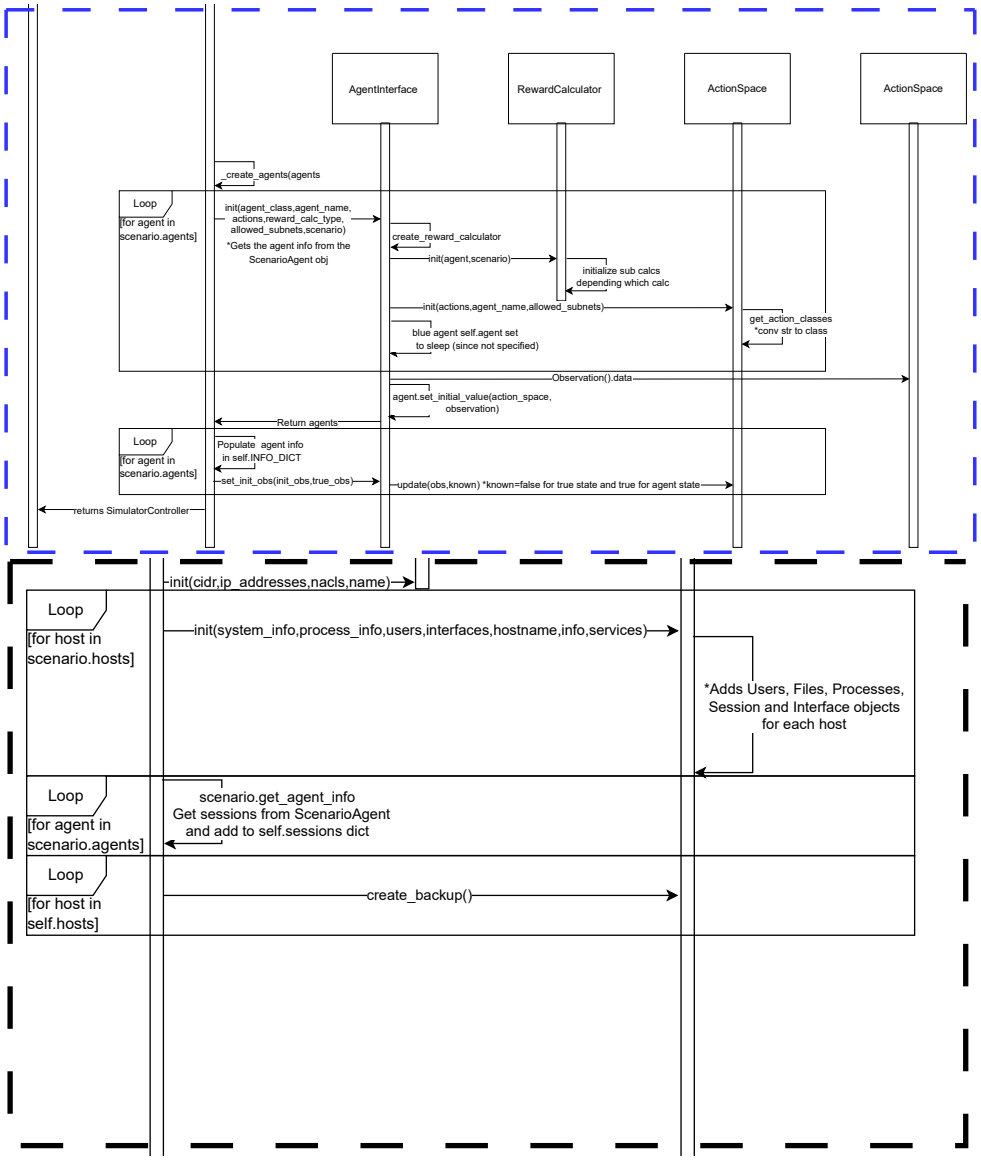


Figure A.1: Sequence diagram illustrating the entities and processes involved when instantiating a new instance of the Cage Challenge 2 environment. Colored boxes are expanded in subsequent plots.


CybORG
SimulationController
(EnvironmentController)
EnvironmentController
Scenario
init(scenario_file,env,agents)
*scenario_file=path to YAML
*env=string ('sim')
*agents=dict of agent objs controlled by env
init(scenario_file,agents)
super.init(scenario_filepath, scenario_mod,agents)
*scenario_filepath=path to YAML
*scenario_mod=default empty dict
*agents=dict of ag objs controlled by env
_parse_scenario(scenario_path)
*Puts YAML into dict using yaml.load
Continue parsing scenario
Updates each host using respective image (each image is own YAML file)
init(scenario_dict)
*scenario_dict is dict that has the YAML info
Loop
[for agent in scenario["Agents"]]
_init(agent_name, agent_info)
Scenario object with all information
_create_environments
ScenarioAgent
ScenarioSession
State
Subnet
(Entity)
Host
(Entity)
Loop
init(session_info
*All init does is set instance var
*Goes through all info in Scenario object and initializes instance variables: subnet_name_to_cidr,ip_addresses, hosts, sessions, session_count,
init(scenario) - scenario object with all info
_initialize_state(scenario)


AgentInterface
RewardCalculator
ActionSpace
ActionSpace
_create_agents(agents
Loop
[for agent in scenario.agents]
init(agent_class,agent_name, actions,reward_calc_type, allowed_subnets,scenario)
*Gets the agent info from the ScenarioAgent obj
create_reward_calculator
init(agent,scenario)
initialize sub calcs depending which calc
init(actions,agent_name,allowed_subnets)
get_action_classes
*conv str to class
blue agent self.agent set to sleep (since not specified)
Observation().data
agent.set_initial_value(action_space, observation)
Return agents
Loop
[for agent in scenario.agents]
Populate agent info in self.INFO_DICT
set_init_obs(init_obs,true_obs)
update(obs,known) *known=false for true state and true for agent state
returns SimulatorController
init(cidr,ip_addresses,nacls,name)
Loop
[for host in scenario.hosts]
init(system_info,process_info,users,interfaces,hostname,info,services)
*Adds Users, Files, Processes, Session and Interface objects for each host
Loop
[for agent in scenario.agents]
scenario.get_agent_info
Get sessions from ScenarioAgent and add to self.sessions dict
Loop
[for host in self.hosts]
create_backup()

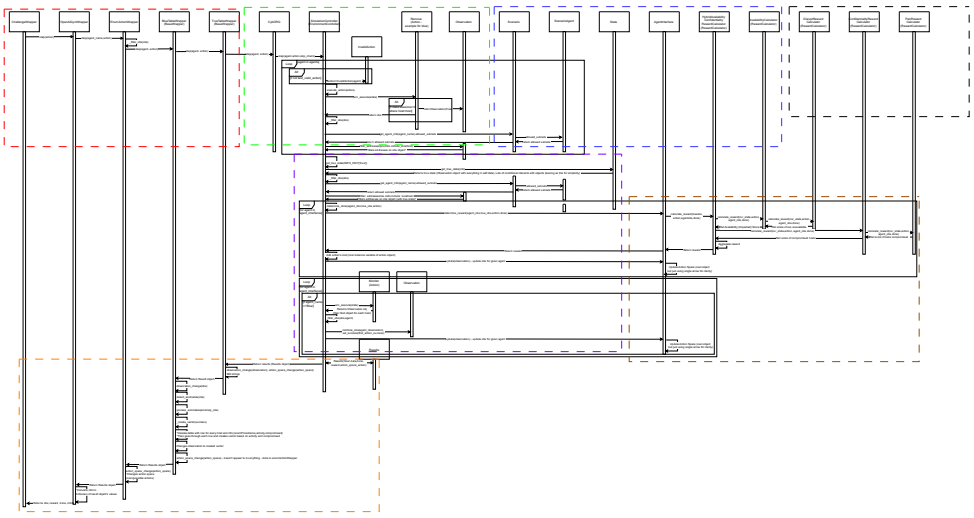
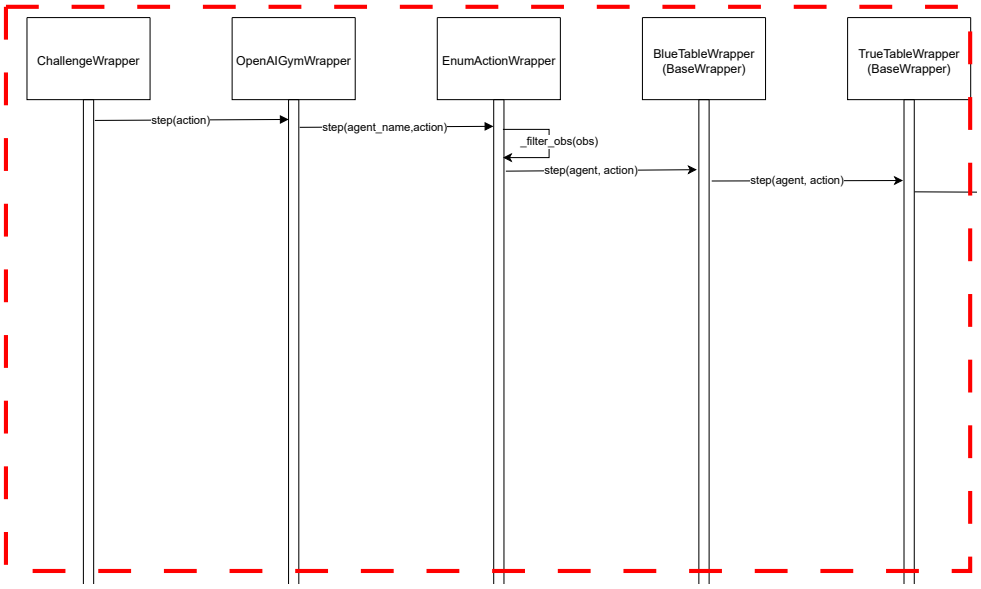
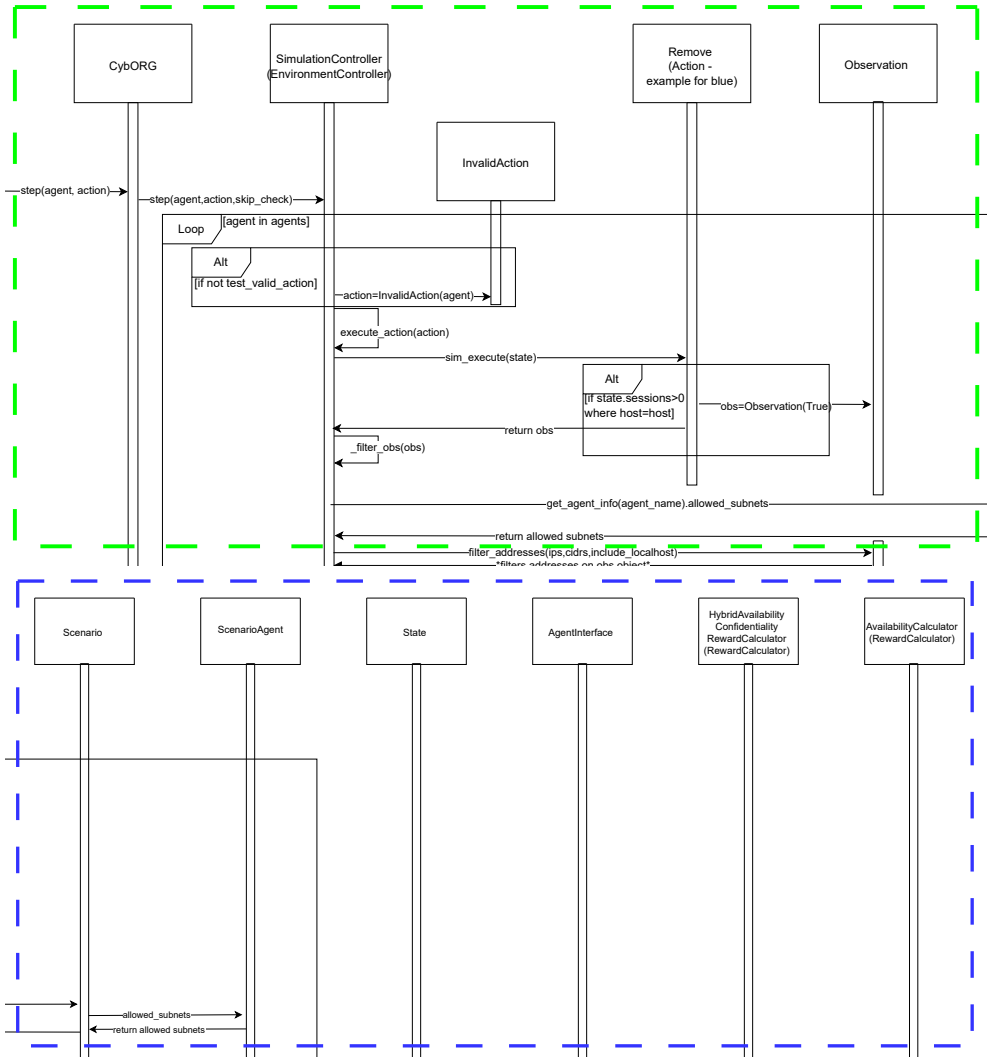
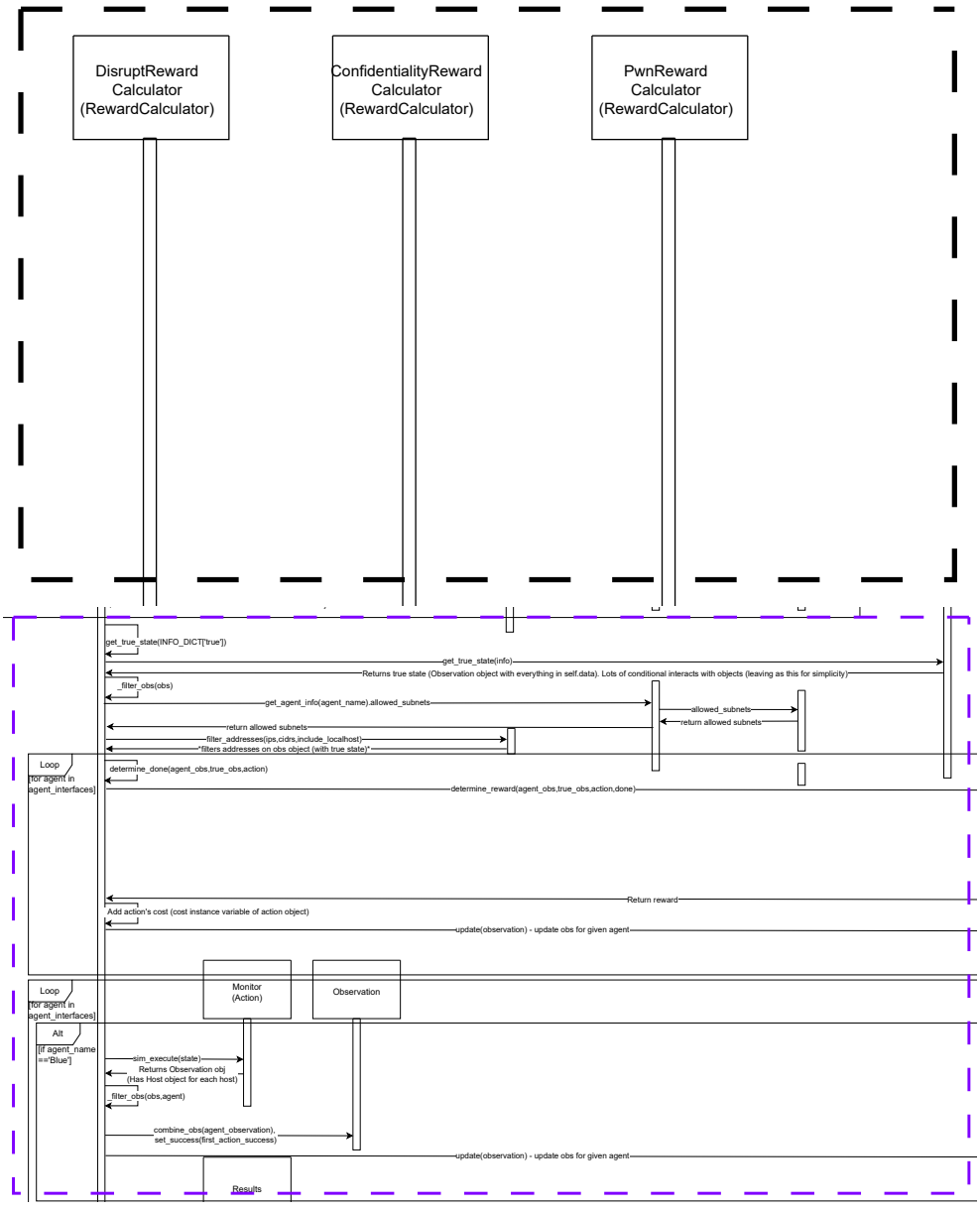
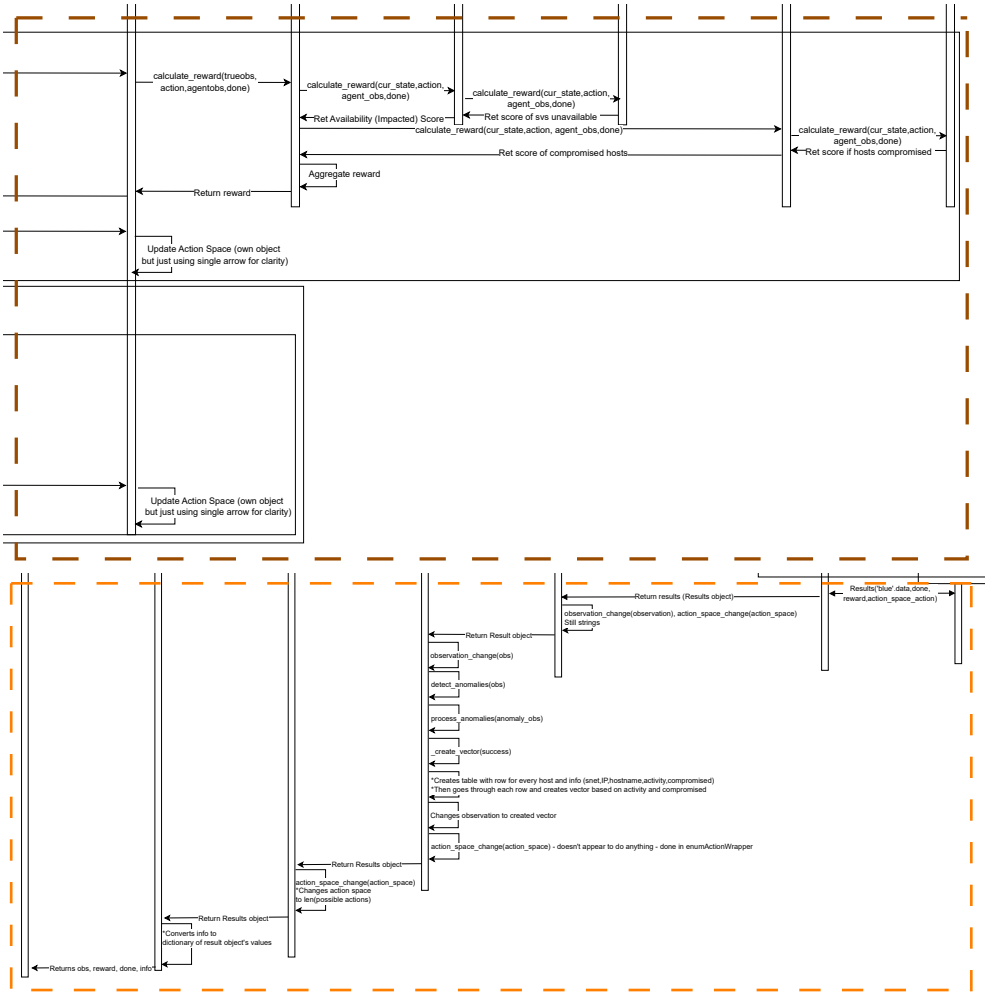


Figure A.2: Sequence diagram illustrating the entities and processes involved every time the environment is interacted with. Colored boxes are expanded in subsequent plots.




CybORG
SimulationController
(EnvironmentController)
Remove
(Action -
example for blue)
Observation
InvalidAction
step(agent, action)
step(agent,action,skip_check)
Loop
[agent in agents]
Alt
[if not test_valid_action]
action=InvalidAction(agent)
execute_action(action)
sim_execute(state)
Alt
[if state.sessions>0
where host=host]
obs=Observation(True)
return obs
_filter_obs(obs)
get_agent_info(agent_name).allowed_subnets
return allowed subnets
filter_addresses(ips,cidrs,include_localhost)
Scenario
ScenarioAgent
State
AgentInterface
HybridAvailability
Confidentiality
RewardCalculator
(RewardCalculator)
AvailabilityCalculator
(RewardCalculator)
allowed_subnets
return allowed subnets


DisruptReward Calculator (RewardCalculator)
ConfidentialityReward Calculator (RewardCalculator)
PwnReward Calculator (RewardCalculator)
get_true_state(INFO_DICT['true'])
get_true_state(info)
Returns true state (Observation object with everything in self.data). Lots of conditional interacts with objects (leaving as this for simplicity)
_filter_obs(obs)
get_agent_info(agent_name).allowed_subnets
allowed_subnets
return allowed subnets
return allowed subnets
filter_addresses(ips,cidrs,include_localhost)
filters addresses on obs object (with true state)
Loop
[for agent in agent_interfaces]
determine_done(agent_obs,true_obs,action)
determine_reward(agent_obs,true_obs,action,done)
Return reward
Add action's cost (cost instance variable of action object)
update(observation) - update obs for given agent
Loop
[for agent in agent_interfaces]
Monitor (Action)
Observation
Alt
[if agent_name =='Blue']
sim_execute(state)
Returns Observation obj (Has Host object for each host)
_filter_obs(obs,agent)
combine_obs(agent_observation), set_success(first_action_success)
update(observation) - update obs for given agent
Results



The selected action and observation space must also be converted to numerical representations that the RL agent can process. The *enumaction* wrapper is used to determine all possible actions, which are then converted to a discrete space for the agent to select from. The *openAIGym* wrapper is used to determine the observation size (feature size of the agent). It does this by initializing the environment and returning the length of the processed observation.

Another important wrapper is the *TrueTableWrapper*, which returns the processed true state of the environment (not the agent's perceived state of the environment). This preprocessing is done using a technique similar to the blue agent's preprocessing.

Figure A.3 shows the initialization of the wrappers that are used to process the CybORG data into numerical representations that can be interpreted by RL agents.

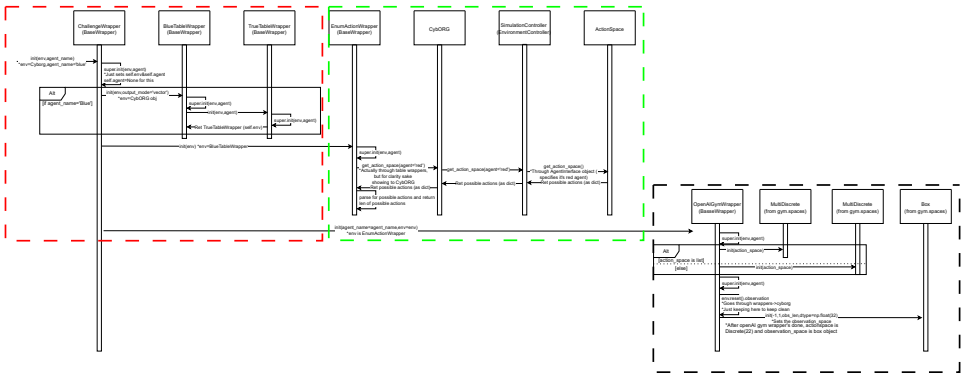
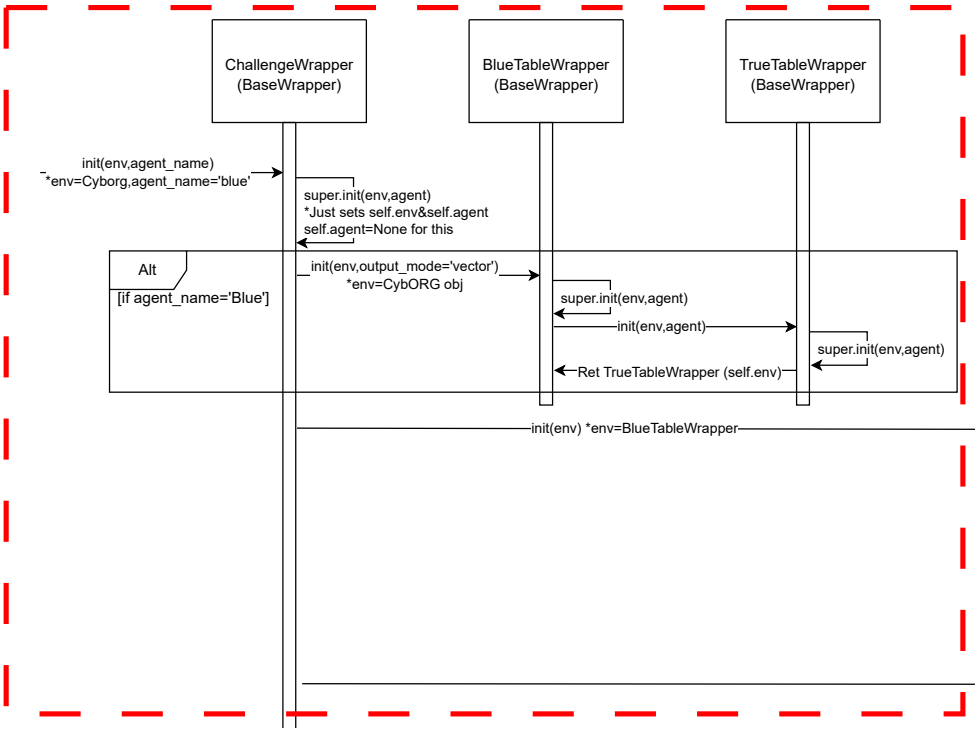
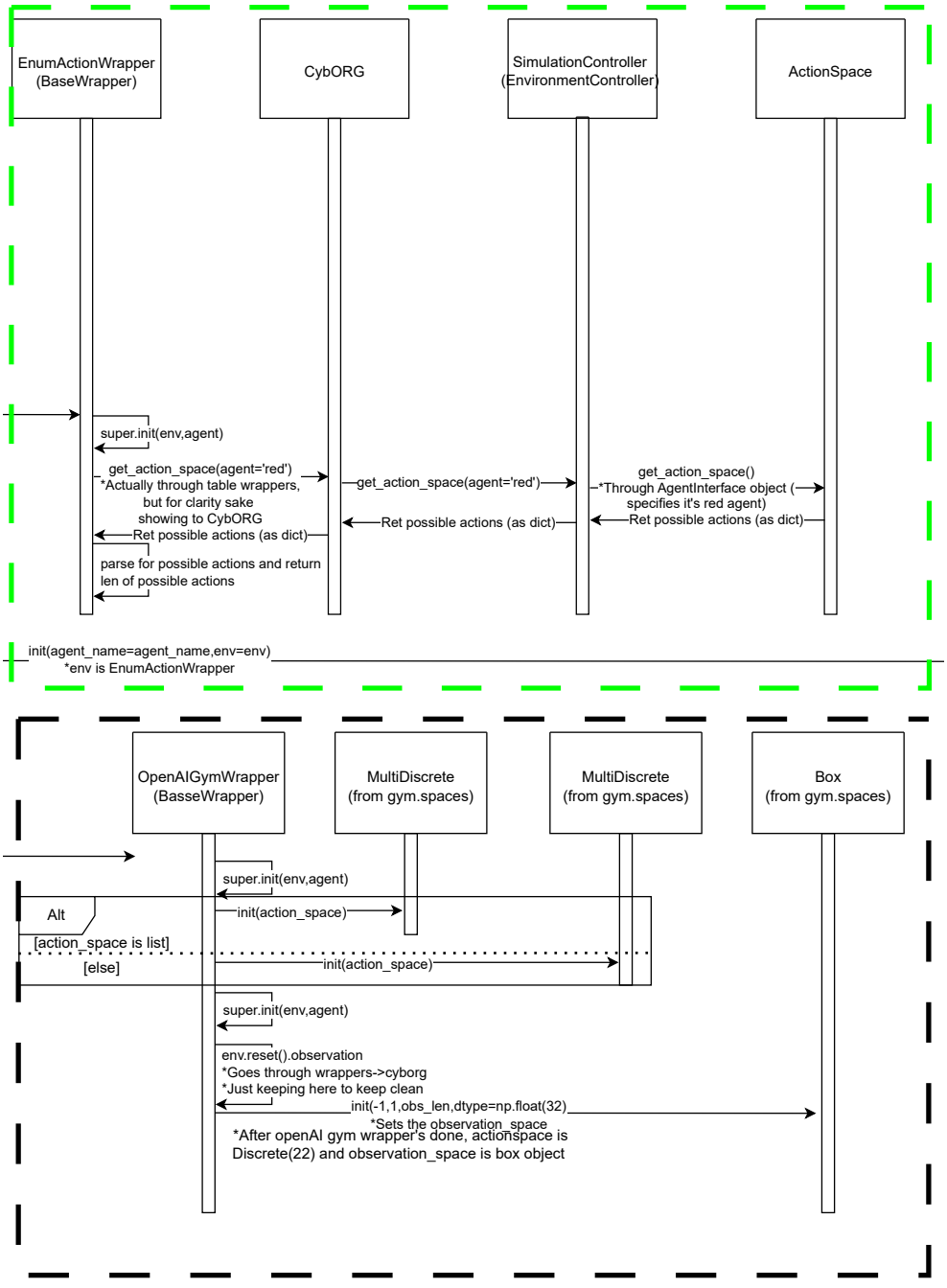


Figure A.3: Sequence diagram illustrating the wrappers that are used to interface between RL agents and the simulated Cage Challenge 2 environment. Colored boxes are expanded in subsequent plots.


ChallengeWrapper
(BaseWrapper)
BlueTableWrapper
(BaseWrapper)
TrueTableWrapper
(BaseWrapper)
init(env,agent_name)
*env=Cyborg,agent_name='blue'
super.init(env,agent)
*Just sets self.env&self.agent
self.agent=None for this
Alt
[if agent_name='Blue']
init(env,output_mode='vector')
*env=CybORG obj
super.init(env,agent)
init(env,agent)
super.init(env,agent)
Ret TrueTableWrapper (self.env)
init(env) *env=BlueTableWrapper


EnumActionWrapper
(BaseWrapper)
CybORG
SimulationController
(EnvironmentController)
ActionSpace
super.init(env,agent)
get_action_space(agent='red')
*Actually through table wrappers,
but for clarity sake
showing to CybORG
Ret possible actions (as dict)
parse for possible actions and return
len of possible actions
get_action_space(agent='red')
Ret possible actions (as dict)
get_action_space()
*Through AgentInterface object (
specifies it's red agent)
Ret possible actions (as dict)
init(agent_name=agent_name,env=env)
*env is EnumActionWrapper
OpenAIGymWrapper
(BasseWrapper)
MultiDiscrete
(from gym.spaces)
MultiDiscrete
(from gym.spaces)
Box
(from gym.spaces)
super.init(env,agent)
Alt
[action_space is list]
init(action_space)
[else]
init(action_space)
super.init(env,agent)
env.reset().observation
*Goes through wrappers->cyborg
*Just keeping here to keep clean
init(-1,1,obs_len,dtype=np.float(32)
*Sets the observation_space
*After openAI gym wrapper's done, actionspace is
Discrete(22) and observation_space is box object

B Django Application for Agent Evaluation

This appendix contains additional information on the functionality of the Django application [80] that was initially used to evaluate environment modifications and agent development.

B.1 Application Overview

The Django application has three main functionalities that are available through its graphical user interface (GUI):

- The ability to specify hyperparameters for a new round, run an iteration, and store the results.
- The analysis of rounds at an episodic granularity.
- The analysis of rounds at a per timestep granularity.

B.2 Running a Round

Games can be directly run from the interface, enabling users to select the number of episodes, steps and batch size. Additional parameters cannot be set through the interface and must instead be configured by modifying the code directly.

The game's associated data, including the actions taken, rewards and states are then stored in the database to be analyzed further. The interface for running games is shown in Figure B.1.

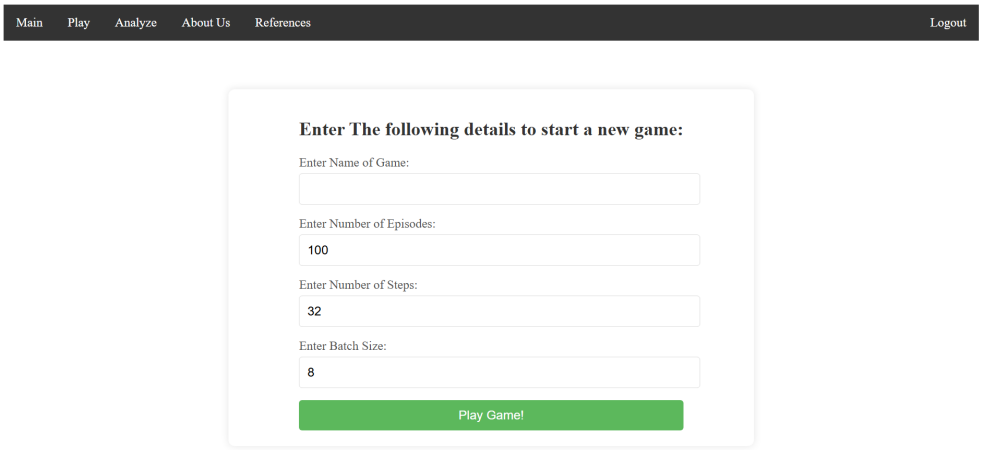


Figure B.1: Django interface to run a game.

B.3 Analyzing Results at the Episodic Level

Once games have been run and stored in the database, they can be analyzed further from the interface. Figure B.2 shows the interface for selecting previously run games for deeper analysis.

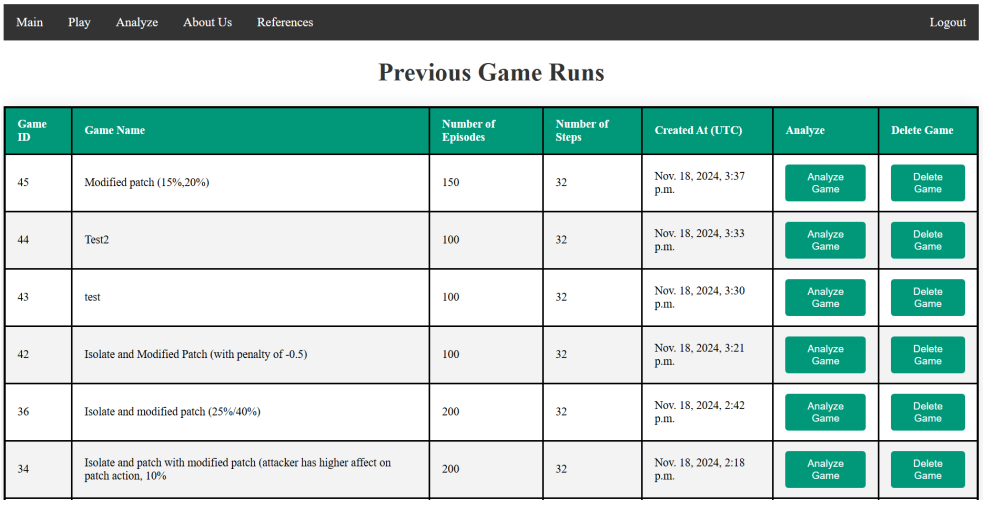


Figure B.2: Interface for analysis selection.

After selecting a game from the interface shown in Figure B.2, an episodic level view for the analysis appears, including:

- All actions taken for that episode; and
- The action taken at each timestep for that episode; and
- The reward across all episodes.

All of these metrics are presented in the form of Chart.js [82] visualizations. The interface allows for different types of charts to be generated depending on the intent. Furthermore, individual episodes can be selected to provide deeper insight into the actions taken during that particular episode. The episodic analysis view is shown in Figure B.3.

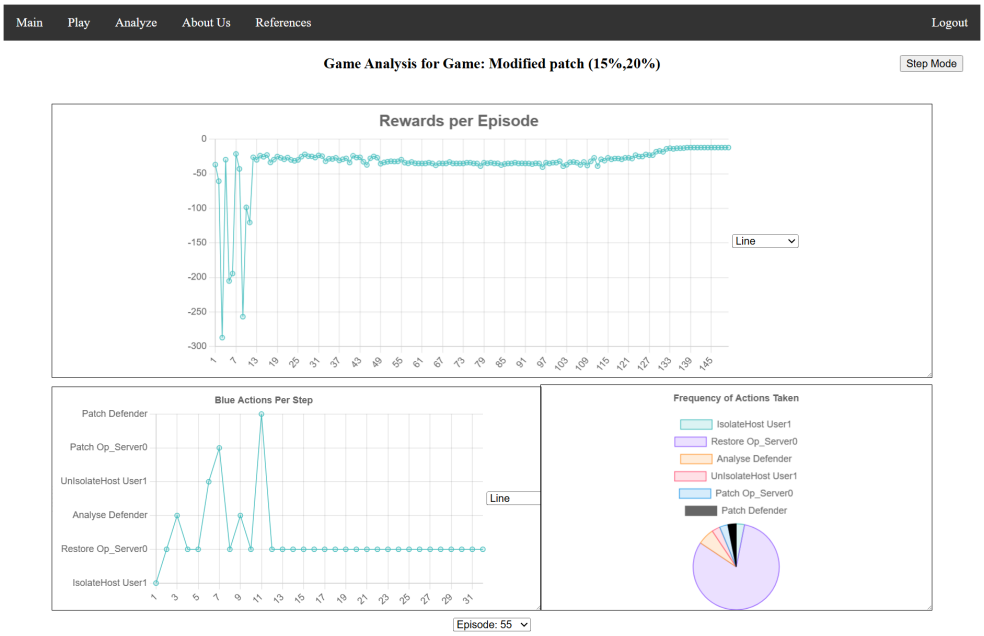


Figure B.3: Illustration of the episodic analysis view for a game.

B.4 Analyzing Results at the Timestep Level

The Django application supports a deeper analysis for the individual timesteps within a game. This view dynamically generates an interface, showing the blue agent's perceived state and the true state of the environment at a given timestep.

In addition to this, the rewards for that particular step, the last blue action, the last red action, the next blue action, and the next red action are shown for each timestep. Figure B.4 shows the analysis of a 4-host and 13-host game using the deeper per-timestep interface.

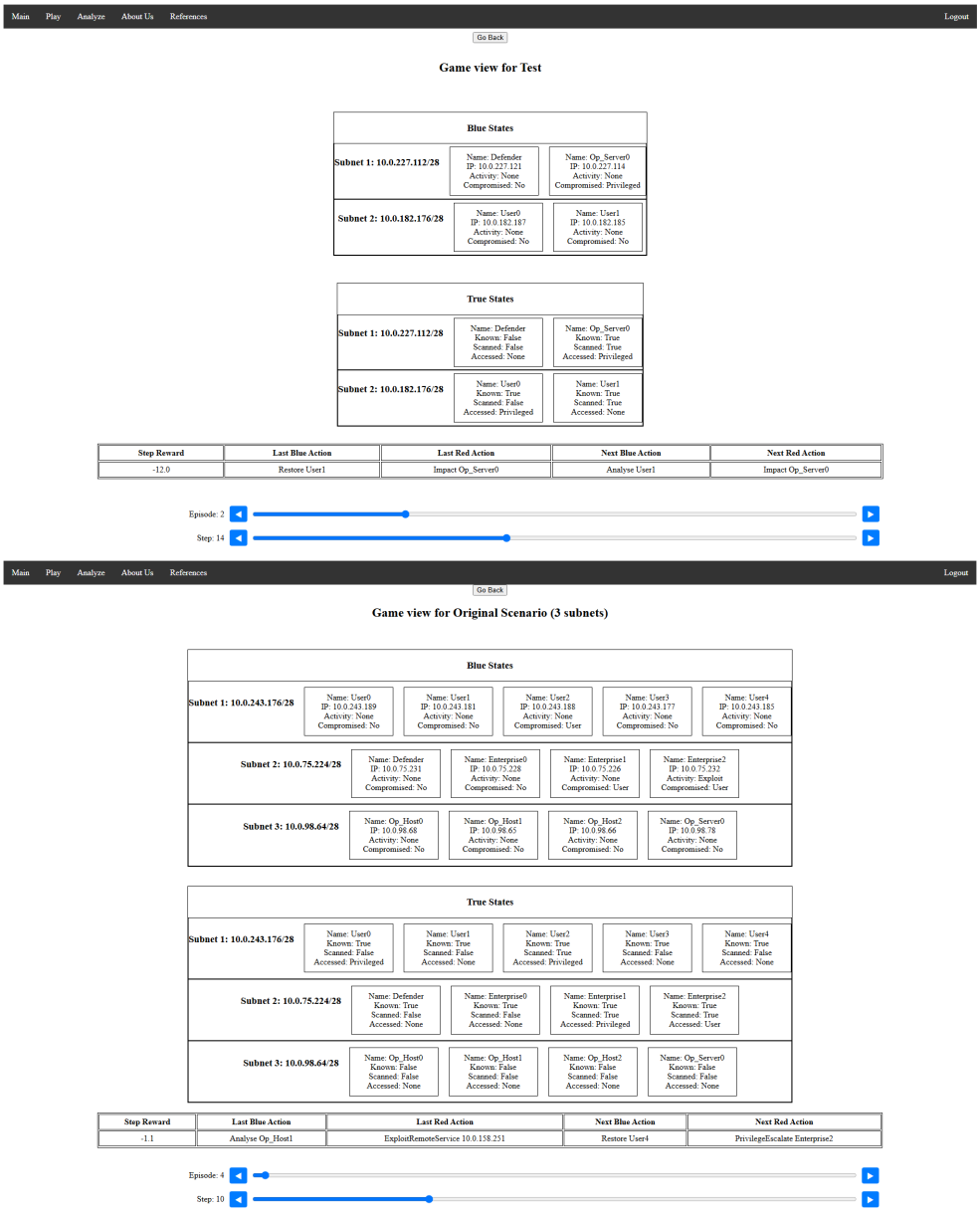


Figure B.4: Illustration of per-timestep analysis of a 4-host (top) and 13-host (bottom) game.

C Prompts used for Study

As discussed, three main prompt formats were used for this study:

- The initial prompt, where the state of each host was listed in JSON format with an explicit priority and last-analyzed status for each.
- The updated prompt that described the priority as a function of hops to the operational server rather than a label. This also included converting the actions and host names into a generic format.
- An optimized prompt that described the steps the LLM should take in deciding an action in an explicit step-by-step format. This prompt included specific constraints, such as only selecting the remove and restore actions on hosts with suspicious processes or files.

C.1 Initial Prompt

The initial prompt was used in Phase 1 to validate the LLMs. It included an isolated, last-analyzed and priority status and listed all the processes, files, and scans that were not in the baseline image. This was all presented in a JSON format surrounded by additional contextual information to enable the LLM to make a more informed decision. The initial prompt format is shown in Listing C.1.

Listing C.1: Initial prompt. Extra newlines have been added to the original to be displayed on the width of a page.

```
You are defending a network at timestep 26. Your goal is to keep
the network fully functional. Choose an action from Restore,
Patch, Isolate, Unisolate, Remove, Analyse, Block, Allow.
Only select actions that disrupt functionality if essential.
Respond with <Action> <Hostname> only. Current network state:
{"Defender": {"ip": "10.0.13.171", "Processes": [], "Files": [],
"Scans": [], "Isolated": false, "LastAnalysed": 0,
"Priority": "High"},
"Enterprise0": {"ip": "10.0.13.164", "Processes": [],
```

```
"Files": [], "Scans": [], "Isolated": false,
"LastAnalysed": 6, "Priority": "Medium"},
"Enterprise1": {"ip": "10.0.13.173", "Processes":
[{"remoteAddress": "10.0.114.134", "localPort": 49547,
"count": 1}], "Files": [{"Name": "cmd.exe", "Path": "C:\\temp\\",
"Density": 0.9, "Signed": false}, {"Name": "escalate.exe",
"Path": "C:\\temp\\", "Density": 0.9, "Signed": false}],
"Scans": [], "Isolated": false, "LastAnalysed": 14,
"Priority": "Medium"},
"Enterprise2": {"ip": "10.0.13.170", "Processes":
[{"remoteAddress": "10.0.114.134", "localPort": 49191,
"count": 1}], "Files": [], "Scans": [], "Isolated": false,
"LastAnalysed": -1, "Priority": "Medium"},
"Op_Host0": {"ip": "10.0.105.6", "Processes": [], "Files": [],
"Scans": [], "Isolated": false, "LastAnalysed": 23,
"Priority": "High"},
"Op_Host1": {"ip": "10.0.105.11", "Processes": [], "Files": [],
"Scans": [], "Isolated": false, "LastAnalysed": 1,
"Priority": "High"}, "Op_Host2": {"ip": "10.0.105.3",
"Processes": [], "Files": [], "Scans": [], "Isolated": false,
"LastAnalysed": 8, "Priority": "High"},
"Op_Server0": {"ip": "10.0.105.7", "Processes":
[{"remoteAddress": "10.0.114.134", "localPort": 22,
"count": 11}], "Files": [], "Scans": [], "Isolated": false,
"LastAnalysed": 18, "Priority": "High"},
"User0": {"ip": "10.0.114.134", "Processes": [], "Files": [],
"Scans": [], "Isolated": false, "LastAnalysed": 13,
"Priority": "Low"},
"User1": {"ip": "10.0.114.129", "Processes":
[{"remoteAddress": "10.0.114.134", "localPort": 56433,
"count": 1}], "Files": [{"Name": "cmd.exe", "Path": "C:\\temp\\",
"Density": 0.9, "Signed": false}], "Scans": [],
"Isolated": false, "LastAnalysed": 3, "Priority": "Low"},
"User2": {"ip": "10.0.114.137", "Processes": [], "Files": [],
"Scans": [], "Isolated": false, "LastAnalysed": -1,
"Priority": "Low"},
"User3": {"ip": "10.0.114.135", "Processes": [], "Files": [],
"Scans": [], "Isolated": false, "LastAnalysed": 7,
"Priority": "Low"},
"User4": {"ip": "10.0.114.138", "Processes": [], "Files": [],
"Scans": [{"remoteAddress": "10.0.114.134", "localPort": 22,
"count": 1}, {"remoteAddress": "10.0.114.134", "localPort": 80,
"count": 1}, {"remoteAddress": "10.0.114.134", "localPort": 3390,
"count": 1}, {"remoteAddress": "10.0.114.134", "localPort": 443,
"count": 1}, {"remoteAddress": "10.0.114.134", "localPort": 25,
"count": 1}], "Isolated": false, "LastAnalysed": 2,
"Priority": "Low"}}
Best Action:
```

C.2 Standard Prompt

The standard prompt was created to rectify some of the performance issues observed with the initial prompt (mainly high variance). It omitted the priority attribute for each host and instead denoted the priority with respect to how many hops each host had from the operational server along the critical paths (the most direct path to the operational server from that particular host). It also omitted the last-analyzed and timestep attributes, as the LLM appeared to have difficulty processing their reinforcement learning specific metrics (both were specified in terms of timesteps).

Furthermore, the LLM appeared to recognize patterns in specific actions and hosts that did not align with the CybORG environment. As such, these actions and hosts were mapped into generic names. For example, in the context of CybORG, the remove action only functions for non-privileged presences; however, the LLM would consistently recommend the remove action on hosts with root-level red access, despite having this constraint defined explicitly in the prompt.

The surrounding context was modified to include a detailed role, objective, action definition, and execution intent. The standard prompt is shown in Listing C.2.

Listing C.2: Standard prompt. Extra newlines have been added to the original to be displayed on the width of a page.

```
**Role**:
You are an expert at cybersecurity and network defense.

**Objective**:
You must choose **ONE ACTION** to execute on the most critical
host to maintain network functionality.

**Prioritize Hosts by Distance to the Target**
- Prioritize hosts that are closest to the operational server:
- host8: 0 hops.
- host4: 1 hops.
- host2: 2 hops.
- host3: 2 hops.
- host10: 3 hops.
- host11: 3 hops.
- host12: 3 hops.
- host13: 3 hops.

**Determine the Best Action**
For the selected host, choose the best action based on the
following definitions:
```

```
- **action1**: Use to gather additional information on a host,
such as active processes, connections, or files.
- **action2**: Use to revert a fully compromised host where the
attacker has gained persistence.
- **action3**: Use to terminate non-persistent malicious
processes before they escalate privileges or gain persistence.
- **action4**:Use to apply security updates and reduce the
likelihood of future exploitation.
- **action5**: Use to block network access to and from a
compromised host to prevent lateral movement.
- **action6**: Use to restore network access to a previously
isolated host once it is deemed safe.

### **Current Network State**:
host8| IP: 10.0.179.17, NOT ISOLATED, Files: [], Processes:
[11 processes with: (Remote IP: 10.0.243.220 and Port: 22)],
Scans: []
host4| IP: 10.0.232.83, NOT ISOLATED, Files:
[cmd.exe at C:\temp\ (Density: 0.9, Signed: No), escalate.exe
at C:\temp\ (Density: 0.9, Signed: No)], Processes: [1 process
with: (Remote IP: 10.0.243.220 and Port: 57984)], Scans: []
host2| IP: 10.0.232.84, NOT ISOLATED, Files: [escalate.sh at
/tmp/ (Density: 0.9, Signed: No)], Processes: [11 processes
with: (Remote IP: 10.0.243.220 and Port: 22)], Scans: []
host3| IP: 10.0.232.93, NOT ISOLATED, Files: [], Processes: [],
Scans: []
host10| IP: 10.0.243.219, NOT ISOLATED, Files: [], Processes: [],
Scans: []
host11| IP: 10.0.243.211, NOT ISOLATED, Files: [], Processes: [],
Scans: [1 scan with: (Remote IP: 10.0.243.220 and Port: 445),
1 scan with: (Remote IP: 10.0.243.220 and Port: 139),
1 scan with: (Remote IP: 10.0.243.220 and Port: 135),
1 scan with: (Remote IP: 10.0.243.220 and Port: 3389)]
host12| IP: 10.0.243.213, NOT ISOLATED, Files: [], Processes: [],
Scans: []
host13| IP: 10.0.243.215, NOT ISOLATED, Files: [cmd.sh at /tmp/
(Density: 0.9, Signed: No)], Processes: [1 process with:
(Remote IP: 10.0.243.220 and Port: 54437)], Scans: []
host1| IP: 10.0.232.87, NOT ISOLATED, Files: [], Processes: [],
Scans: []
host5| IP: 10.0.179.27, NOT ISOLATED, Files: [], Processes: [],
Scans: []
host6| IP: 10.0.179.20, NOT ISOLATED, Files: [], Processes: [],
Scans: []
host7| IP: 10.0.179.18, NOT ISOLATED, Files: [], Processes: [],
Scans: []
host9| IP: 10.0.243.220, NOT ISOLATED, Files: [], Processes: [],
Scans: []
```

```
Your response must follow this format exactly:
<action> <hostname> (e.g., actionx hostx).

Best Action:
```

The optimized prompt was created to achieve the best performance in CybORG by providing the LLM with step-by-step instructions on how to process the environment's state, and provide a relevant response. This is a similar idea to chain-of-thought prompting [74]; however, instead of encouraging the LLM to follow a step-by-step sequence to produce a response, these steps are explicitly provided.

In addition to this step-by-step methodology, explicit constraints were specified in the prompt, such as disallowing the remove or restore action on hosts without suspicious processes or files. ChatGPT-4o [38] was leveraged to help format these constraints in a way that the intent could easily be extracted by an LLM.

The optimized prompt used for this study is shown in Listing C.3.

C.3 Optimized Prompt

Listing C.3: Optimized prompt. Extra newlines have been added to original to be displayed on width of page

```
**Role**:
You are an expert at cybersecurity and network defense.

**Objective**:
You must choose **ONE ACTION** to execute on the most critical
host to maintain network functionality.

**Instructions**:
Evaluate the Network State and follow these strict steps:

### **Step 1: Identify Critical Hosts**
!!! **IMPORTANT:**
- **DO NOT select action2 or action3 unless there are suspicious
processes or files.**
- Identify hosts with **suspicious processes or files**.

### **Step 2: Prioritize Hosts by Distance to the Target**
- Prioritize hosts that are closest to the operational server:
- host8: 0 hops.
- host4: 1 hops.
- host2: 2 hops.
- host3: 2 hops.
```

```
- host10: 3 hops.
- host11: 3 hops.
- host12: 3 hops.
- host13: 3 hops.

### **Step 3: Select the Best Host**
- **host8 can ONLY be selected if it has suspicious processes or
files.**
- **IF AND ONLY IF** host8 has **ANY suspicious processes or
files**, select host8.
- **IF host8 has NO suspicious processes or files, SKIP IT and
proceed to Step 4.**

### **Step 4: Choose the closest host from the remaining list
that has suspicious processes or files.**

### **Step 5: Determine the Best Action**
For the selected host, choose the best action based on the
following rules:
- **action1**: Use when a host's **status is unknown or
unclear**, and additional information is needed to make a
decision.
- **action2**: Use **ONLY** if the host has suspicious files or
long running processes.
- **action3**: Use **ONLY** if the host has short running
processes.
- **action4**: Use if the host has no current suspicious
processes or files to prevent future exploitation.
- **action5**: Use if the host is NOT already isolated and has
suspicious processes or files to prevent spreading to host8.
- **action6**: Use if the host is already isolated AND has no
suspicious processes or files.

!!! **IP addresses are identifiers only. They must NOT be used
for decision-making. The correct action is determined ONLY by the
host's state and proximity (i.e., number of hops).** !!!
!!! **DO NOT output explanations or repeat instructions. The
correct action is determined ONLY by the host's state and
proximity (i.e., number of hops).** !!!
!!! **ONLY SELECT action2 or action3 if there are suspicious
processes or files.** !!!
!!! **If the host has NO suspicious processes or files ("Files:
[]", "Processes: []"), action2 and action3 are NOT ALLOWED.** !!!

### **Current Network State**:
host8| IP: 10.0.108.129, NOT ISOLATED, Files: [], Processes: [11
processes with: (Remote IP: 10.0.146.86 and Port: 22)], Scans: []
host4| IP: 10.0.16.133, NOT ISOLATED, Files: [cmd.exe at
C:\temp\ (Density: 0.9, Signed: No), escalate.exe at
```

```
C:\temp\ (Density: 0.9,Signed: No)], Processes: [1 process with:
(Remote IP: 10.0.146.86 and Port: 55497)], Scans: []
host2| IP: 10.0.16.134, NOT ISOLATED, Files: [], Processes: [],
Scans: []
host3| IP: 10.0.16.131, NOT ISOLATED, Files: [], Processes: [1
process with: (Remote IP: 10.0.146.86 and Port: 51248)],
Scans: []
host10| IP: 10.0.146.93, NOT ISOLATED, Files: [], Processes: [],
Scans: []
host11| IP: 10.0.146.90, NOT ISOLATED, Files: [cmd.exe at
C:\temp\ (Density: 0.9, Signed: No), escalate.exe at C:\temp\
(Density: 0.9, Signed: No)], Processes: [1 process with:
(Remote IP: 10.0.146.86 and Port: 56133)], Scans: []
host12| IP: 10.0.146.85, NOT ISOLATED, Files: [], Processes: [],
Scans: [1 scan with: (Remote IP: 10.0.146.86 and Port: 80),
1 scan with: (Remote IP: 10.0.146.86 and Port: 3389),
1 scan with: (Remote IP: 10.0.146.86 and Port: 443),
1 scan with: (Remote IP: 10.0.146.86 and Port: 25)]
host13| IP: 10.0.146.87, NOT ISOLATED, Files: [], Processes: [],
Scans: []
host1| IP: 10.0.16.138, NOT ISOLATED, Files: [], Processes: [],
Scans: []
host5| IP: 10.0.108.131, NOT ISOLATED, Files: [], Processes: [],
Scans: []
host6| IP: 10.0.108.134, NOT ISOLATED, Files: [], Processes: [],
Scans: []
host7| IP: 10.0.108.137, NOT ISOLATED, Files: [], Processes: [1
process with: (Remote IP: 10.0.146.86 and Port: 22)], Scans: []
host9| IP: 10.0.146.86, NOT ISOLATED, Files: [], Processes: [],
Scans: []

Your response must follow this format exactly:
<action> <hostname> (e.g., actionx hostx).

Best Action:
```

D Encoder-Only LLMs

Unlike the decoder-only LLMs that process text auto-regressively in a sequential fashion, encoder-only LLMs process the text bidirectionally, weighing tokens agnostic of their position in the input sequence. This shows promise for this study, where the order in which hosts appear should not impact the LLM's decision. For example, just because the enterprise1 host appears before the enterprise2 host in the prompt, the LLM should not inherently prioritize enterprise1, and instead make its decision solely based on the state and attributes of each host.

This study attempted to integrate an encoder-only LLM into CybORG, specifically SecureBERT [90], a widely used model fine-tuned on cybersecurity data. However, this approach encountered several challenges including:

- The window size for SecureBERT is 512 bytes, greatly limiting the amount of information that can be included in a prompt.
- SecureBERT appeared to struggle with conceptualizing the state of the environment.

Figure D.1 demonstrates the performance deficiencies of SecureBERT in the context of CybORG. SecureBERT outputs a response not relevant to CybORG despite the state only containing two hosts, with one containing an escalate.exe and stuxnet.exe file with high density scores. Furthermore, even when one host is explicitly listed as clean and the other as infected, SecureBERT still does not recommend to act on the correct one with respect to CybORG.

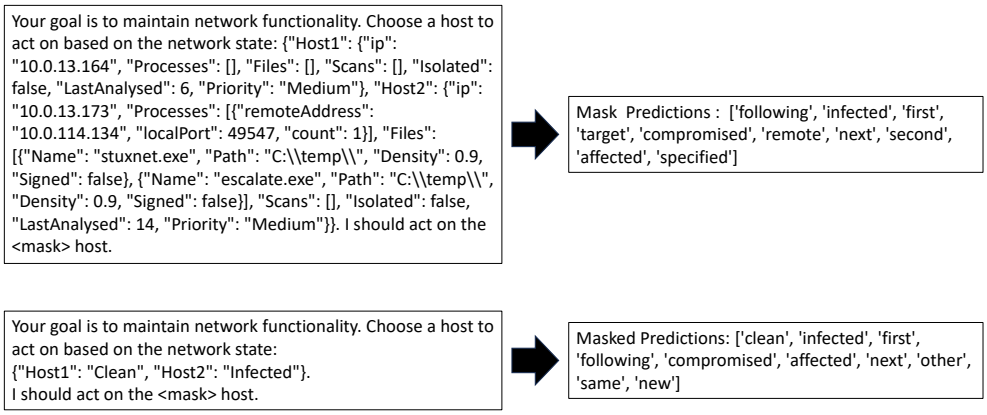


Figure D.1: Deficiencies observed using SecureBERT for CybORG. The top plot shows SecureBERT's output when being fed the state for a simple two-host network, with one host containing obviously malicious files. The bottom plot shows SecureBERT's output when being fed explicit "Clean" and "Infected" labels for each host. In both instances, SecureBERT's response is not favorable in the context of CybORG.

E Mapping the LLM to a Distribution

The incorporation of the teacher's guidance through an auxiliary loss signal demonstrated a positive impact on the agent's training in this study; however, it had a fundamental limitation. Recommending a single action resulted in the agent converging onto a peaked policy where the likelihood of sampling any other action was very unlikely. This made improving beyond this point impossible without first flattening out the distribution, enabling the baseline (independent RL agent) to match or even surpass the teacher-guided agent's performance.

The solution to this problem was to map the LLM's output into a probability distribution and use this entire distribution as part of the loss signal to guide the agent's learning. This resulted in the agent mimicking an entire distribution rather than greatly inflating single probabilities corresponding to the teacher's recommendation.

The process of mapping the LLM's output to a distribution involves analyzing the tokens that correspond to the LLM's recommended action and host to extract their corresponding softmax distributions. Once these distributions are extracted, the probabilities for the tokens corresponding to every action and host are parsed into their own distributions, only including tokens relevant to CybORG (instead of a distribution over every possible token).

From here, simple cartesian multiplication was applied between the action and host distributions to generate probabilities for selecting every action on every host - matching CybORG's action space. This was normalized and used as the LLM's output to guide learning.

The entire process for mapping the LLM's output into a distribution over CybORG's action space is shown in Figure E.1.

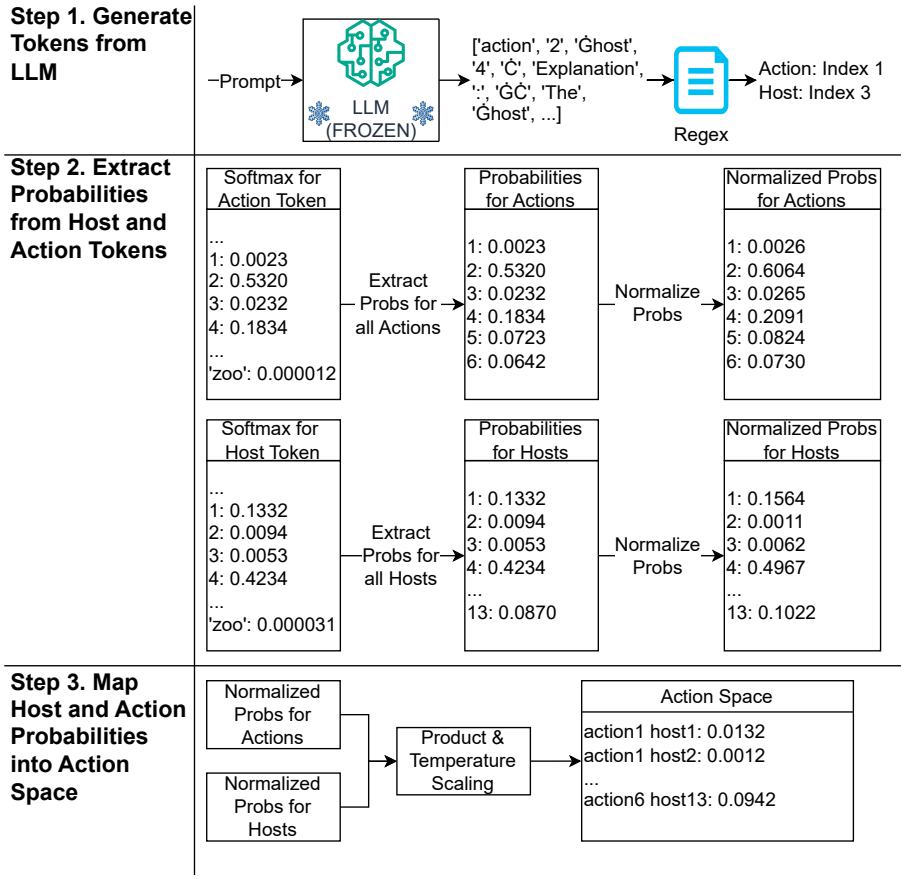


Figure E.1: Extracting a probability distribution from the LLM. In Step 1, the tokens generated by the LLM are parsed and the one representing the recommended action and host are extracted. In Step 2, the softmax for the LLM's predicted action and host is used to extract probabilities for every action and host in the environment. These are then normalized to form a probability distribution across the actions and a probability distribution across the hosts. The 'zoo' token is meant to illustrate that the LLM outputs a distribution across every possible token. In Step 3, the probabilities of the individual actions and hosts are merged to form a distribution across every action on every host (the same action space used for the RL agent). Temperature scaling is used to sharpen the probability distribution (i.e., push probabilities closer to 0 or 1).

The same auxiliary loss process used in this study relies on a single action and cannot be used with the LLM outputting a distribution. As such, the loss

term was modified to use Kullback-Leibler (KL) divergence, a loss function that quantifies the difference between two probability distributions [91]. KL divergence includes information about the entire distribution, enabling the RL agent to modify its policy in a way that aligns with the LLM's full set of recommendations, rather than just optimizing for a single action.

One potential concern observed with the LLM's distribution was the low probability of selecting the highest action compared to all others. As shown in Table E.1, while the highest action has a 24.83% of being taken, there is a 75.17% that it will not be. To rectify this, a temperature scaling of 0.5 was applied to the distribution. Temperature scaling is a technique for tuning the sharpness of a model's distribution, with a scalar T:

$$softmax(z_i) = \frac{e^{z_i/T}}{\sum_{j=1}^{N} e^{z_j/T}}$$

where $T < 1$ results in a sharper distribution, and $T > 1$ yields a flatter distribution.

Table E.1: Illustrating the impact that a temperature scaling of 0.5 has on the probabilities. Probabilities keep their respective ordering; however, are nudged higher or lower depending on their rankings.

	No Temperature	0.5 Temperature
Largest Prob	24.83%	61.19%
Second Largest Prob	11.74%	13.71 %
Third Largest Prob	8.08%	6.48%

E.1 Optimized Prompt

The results of the LLM integration using a distribution instead of a single action for the optimized prompt are shown in Figure E.2. The temperature scaled and unmodified distributions are shown. The hyperparameters are identical to those used in Chapter 5.

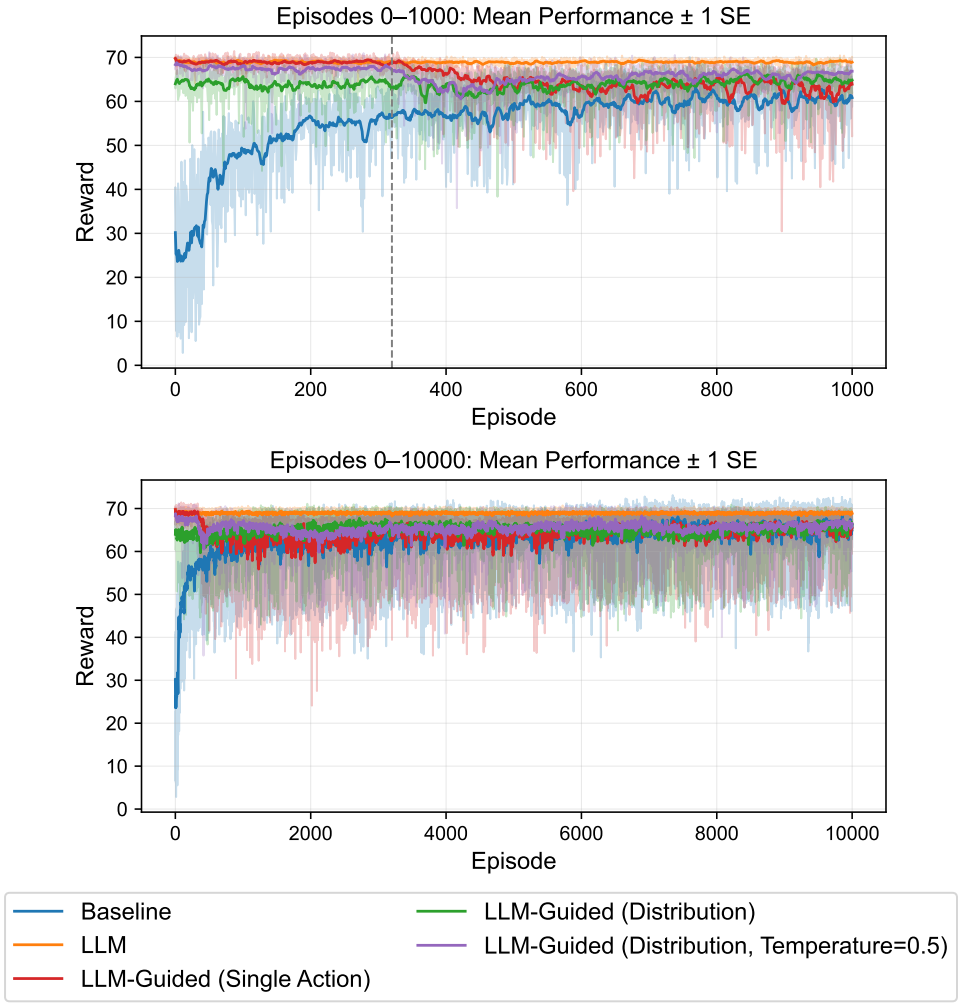


Figure E.2: Demonstrating the performance of LLM integration using a distribution for the optimized prompt. Results show the temperature-scaled and unmodified distribution. The hyperparameters are identical to those used in Chapter 5. The vertical dashed line indicates the point at which the teacher-guided agents have transitioned to fully independent RL (i.e., learning solely from the environment's signals). For clarity, the dashed line is only shown on the top plot.
Top: Mean reward after applying a 10-episode running average with a ±1 standard error for 500 episodes.
Bottom: Mean reward after applying a 10-episode running average with a ±1 standard error for 10,000 episodes.

Figure 5.22 shows a minor decrease in performance for the

recommendation as a distribution during the action masking phase of training. This is expected because, unlike the LLM's default behavior of greedily selecting the token with the highest probability as was done in the single-action method, the distribution method does not guarantee that the LLM's recommendation will be chosen. Because temperature scaling increases the chances of sampling the most likely action, its performance is initially very similar to the single action method.

Both methods show a decrease in performance during the transition to independent RL, but the guidance as a distribution does not immediately revert back to the baseline performance, as is observed with the single-action recommendation. The distribution-based approach yields slightly higher performance until episode ≈4,800, at which point the baseline catches up.

It can also be seen that the standard error for the guidance as a distribution is considerably lower than the single-action recommendation, which is indicative of a more stable teacher-guided implementation.

The actor loss, explained variance, critic loss, and entropy of the distribution-guided and single-action-guided agents are shown in Figure E.3.

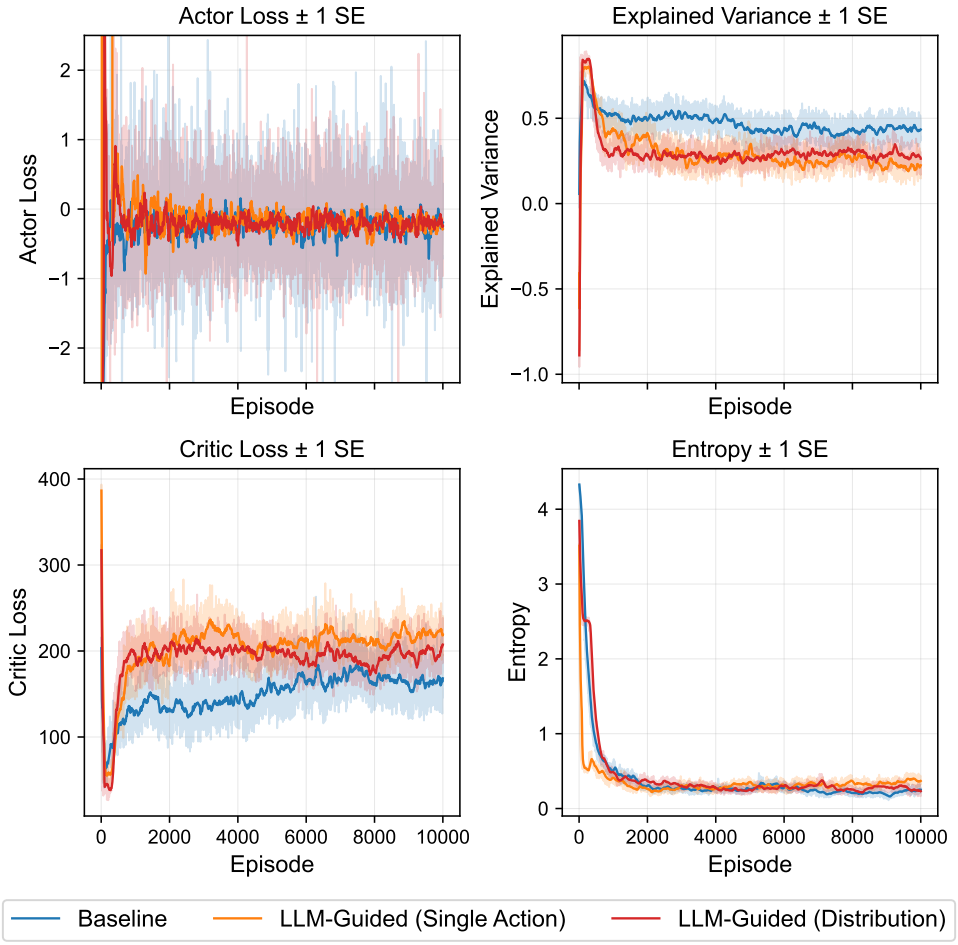


Figure E.3: Comparing the actor loss, explained variance, critic loss, and entropy of the distribution-guided RL agent against the single-action-guided one. The unmodified distribution is used for this comparison.

Figure E.3 shows a minor improvement with respect to explained variance and critic loss for the distribution-guided agent. This can be attributed to the critic being exposed to more states because of the stochastic sampling rather than choosing the most likely token. However, these metrics are both considerably lower than the baseline agent (after transitioning to independent RL). The critic learning solely from the LLM recommendations is the likely cause for these results.

The actor loss is roughly the same between both teacher-guided implementations, showing quick convergence after the transition to

independent RL.

The entropy for the distribution-guided agent closely resembles that of the baseline, likely due to the stochastic nature of distributions.

Overall, these metrics show an improvement with the distribution-guided agent compared to the single-action guided one. The main issue appears to be an unstable critic network, likely attributable to the exploitation of LLM recommendations early on, leading to inadequate training on all states.

E.2 Standard Prompt

The exact same process of mapping the LLM's output into a distribution was also done for the standard prompt. The results of guiding the agent using a distribution instead of a single action are shown in Figure E.4. Like above, the same hyperparameters in Chapter 5 were used for both implementations with a temperature-scaled and modified distribution.

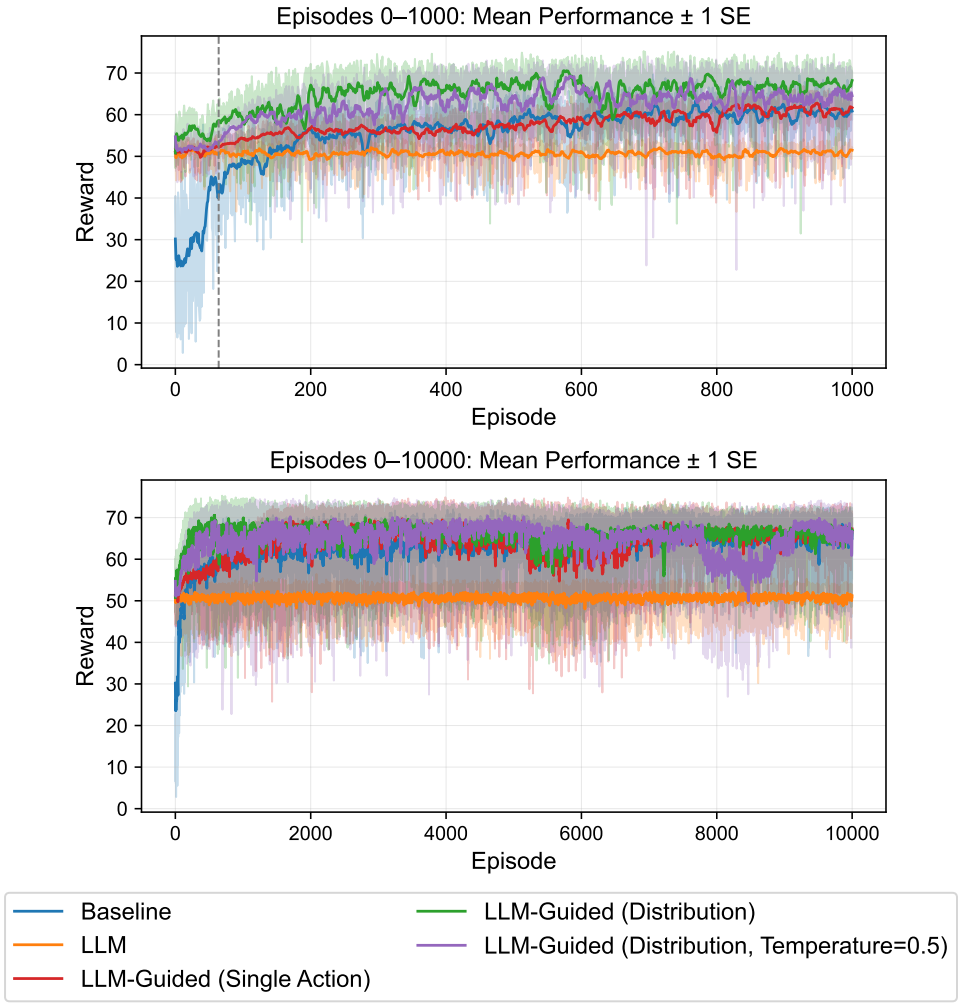


Figure E.4: Demonstrating the performance of LLM integration using a distribution for the standard prompt. Comparison of a temperature-scaled and unmodified distribution against the single-action method used in Chapter 5. The vertical dashed line indicates the point at which the teacher-guided agents have transitioned to fully independent RL (i.e., learning solely from the environment's signals). For clarity, the dashed line is only shown on the top plot.
Top: Mean reward after applying a 10-episode running average with a ±1 standard error for 500 episodes.
Bottom: Mean reward after applying a 10-episode running average with a ±1 standard error for 10,000 episodes.

The recommendation as a distribution shows a considerable increase in

training efficiency compared to the recommendation as a single-action after the transition to independent RL at episode 64.

The initial performance of the unmodified distribution outperforms the temperature-scaled one (and the baseline LLM performance), illustrating the potential benefits of sampling from the LLM's distribution rather than picking the token with the highest probability.

The distribution-guided RL agents outperform the others until episode ≈5,100, where performance becomes comparable to the baseline's.

The temperature-scaled distribution shows noticeable instability and even fluctuates below the baseline agent's performance from episodes ≈7,800-9,100.

Figure E.2 compares the actor loss, explained variance, critic loss, and entropy between the two teacher-guided implementations.

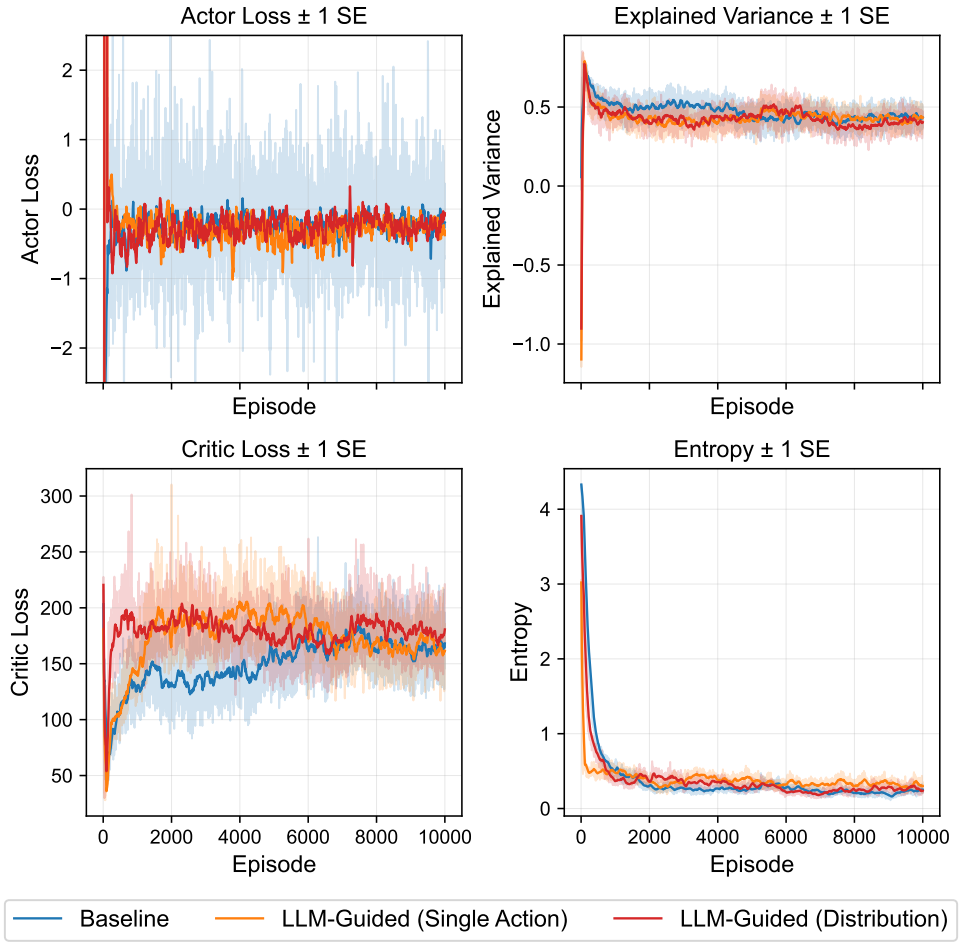


Figure E.5: Comparing the actor loss, explained variance, critic loss, and entropy of the distribution-guided RL agent against the single-action-guided one for the standard prompt. The unmodified distribution is used for this comparison.

Similar to Figure E.3, the distribution-guided implementations using the standard prompt shows comparable critic loss, explained variance, and actor loss metrics with a slight increase in entropy that matches the baseline's behavior.

E.3 Discussion

While mapping the LLM's guidance to a distribution alleviates the primary concern of the distilled RL agent having too narrow of a policy, this does not

fully address the issue with the critic learning solely from LLM recommendations. Similar "warm-starting" techniques used in Figure 5.24 where the critic gets additional training on the agent's rollouts during the transition period, could help the critic produce better value estimates, facilitating a smoother transition to independent RL. This should ultimately reduce the deficiencies in the explained variance and critic loss seen above.

F Evaluating LLM-Integration for Different Scenarios

To demonstrate the LLM's effectiveness across different environments, its impact in the RL process was tested on nine additional scenarios ranging from four to twelve hosts. The action space and functionality are practically identical to the 13-host network used throughout this study, with minor modifications to the red agent and scenarios.

The LLM-guided method is identical to what was used in the study for the standard and optimized prompt, with varying times at which to start the decay, to scale with the complexities of the scenarios. The rate of decay for the auxiliary loss and action mask is the same as in the 13-host network evaluated in Chapter 5, but the time in which the decay starts varies between the scenarios.

The structure of the prompts used for the LLM is almost identical, with the only difference being the hosts and their associated distance from the operational server.

Table F.1: The only hyperparameter adjusted for testing the implementation is when the transition from teacher-guided to independent RL is initiated.

Number of Hosts	**Standard Decay Start**	**Optimized Decay Start**
4	1	10
5	1	10
6	1	10
7	1	15
8	2	20
9	2	20
10	2	20
11	3	25
12	3	30

Overall, the results in all the environments show what was exhibited in the rest of the study - the RL agent quickly converges to the LLM's performance, before stabilizing with the baseline PPO agent. It can be seen that the LLM is less effective and stable in some environments. For example, the baseline performance of the LLM exhibits noticeably high variance in the six-host scenario.

The results of the LLM integration for the standard and optimized prompt are shown in Figures F.1 to F.9.

These results are only for the auxiliary loss and action masking technique employed in Chapter 5, where the LLM recommends a single-action. The recommendation as a distribution technique discussed in Appendix E was not tested for these scenarios.

The top plot and bottom plot for each figure show the mean reward after applying a 10-episode running average with a ± 1 standard error for 500 and 5,000 episodes respectively.

The vertical dashed lines indicate the point at which the teacher-guided agents have transitioned to fully independent RL (i.e., learning solely from the environment's signals). The first dashed line is for the standard prompt (green curve) and the second dashed line is for the optimized prompt (purple curve). For clarity, the dashed lines are only shown on the top plot.

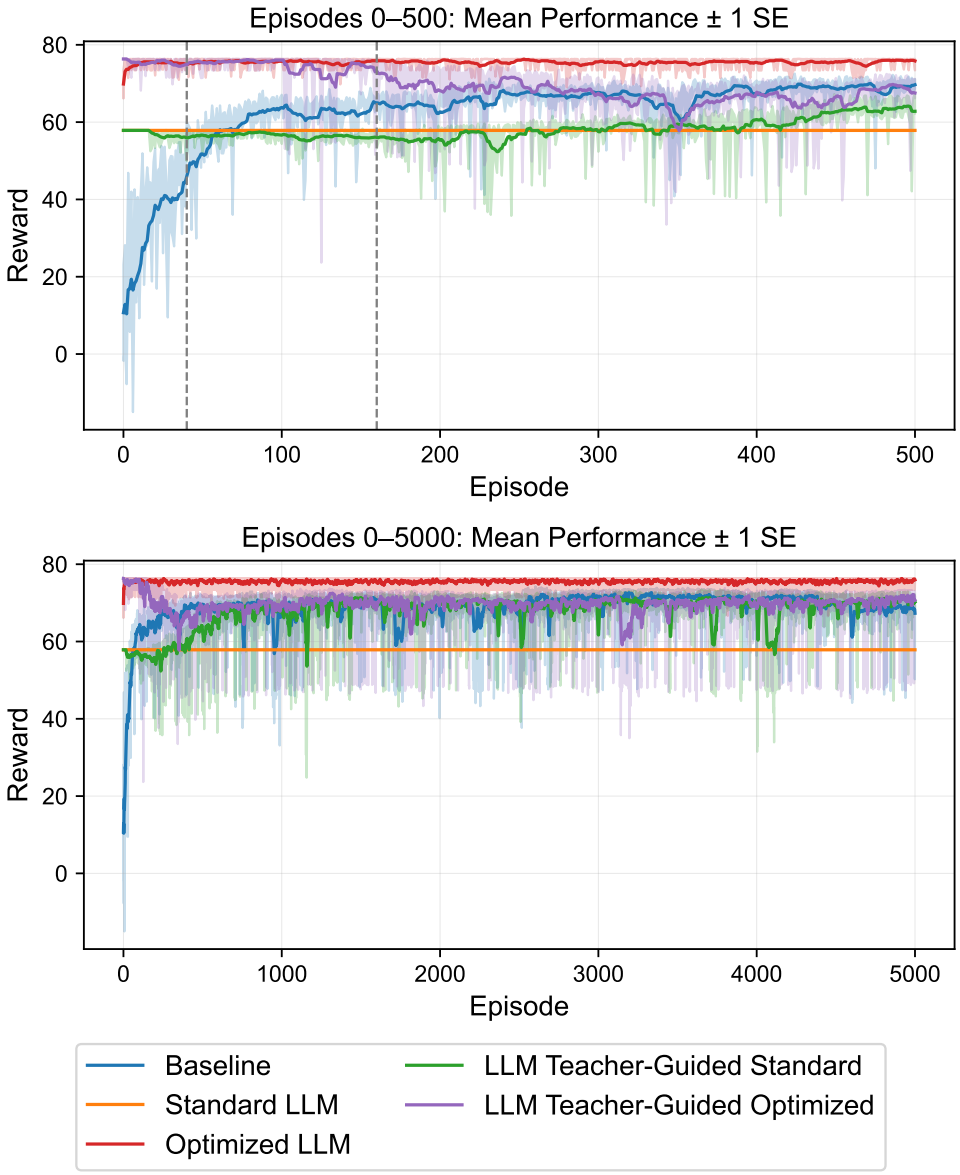


Figure F.1: LLM integration results in the 4-host environment.

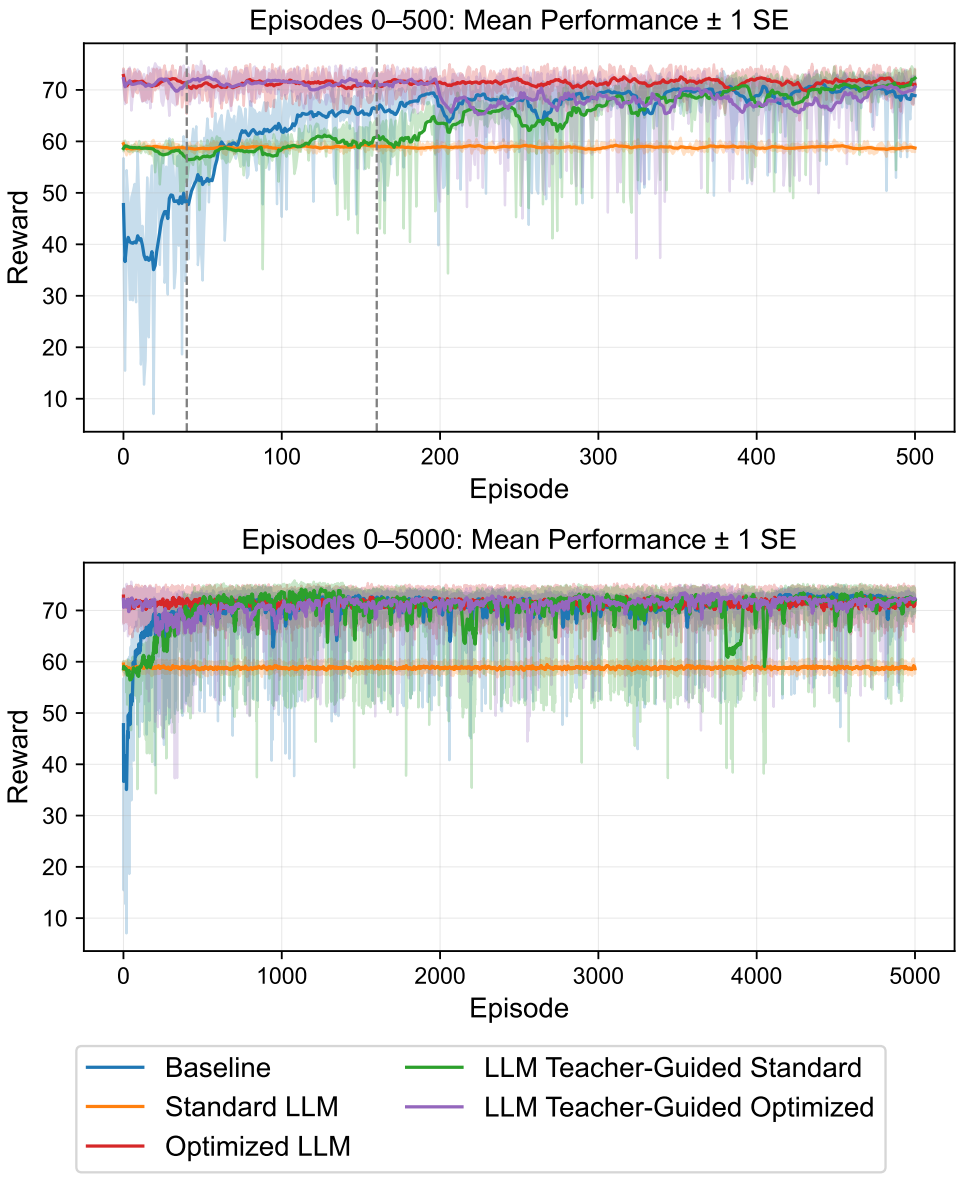


Figure F.2: LLM integration results in the 5-host environment.

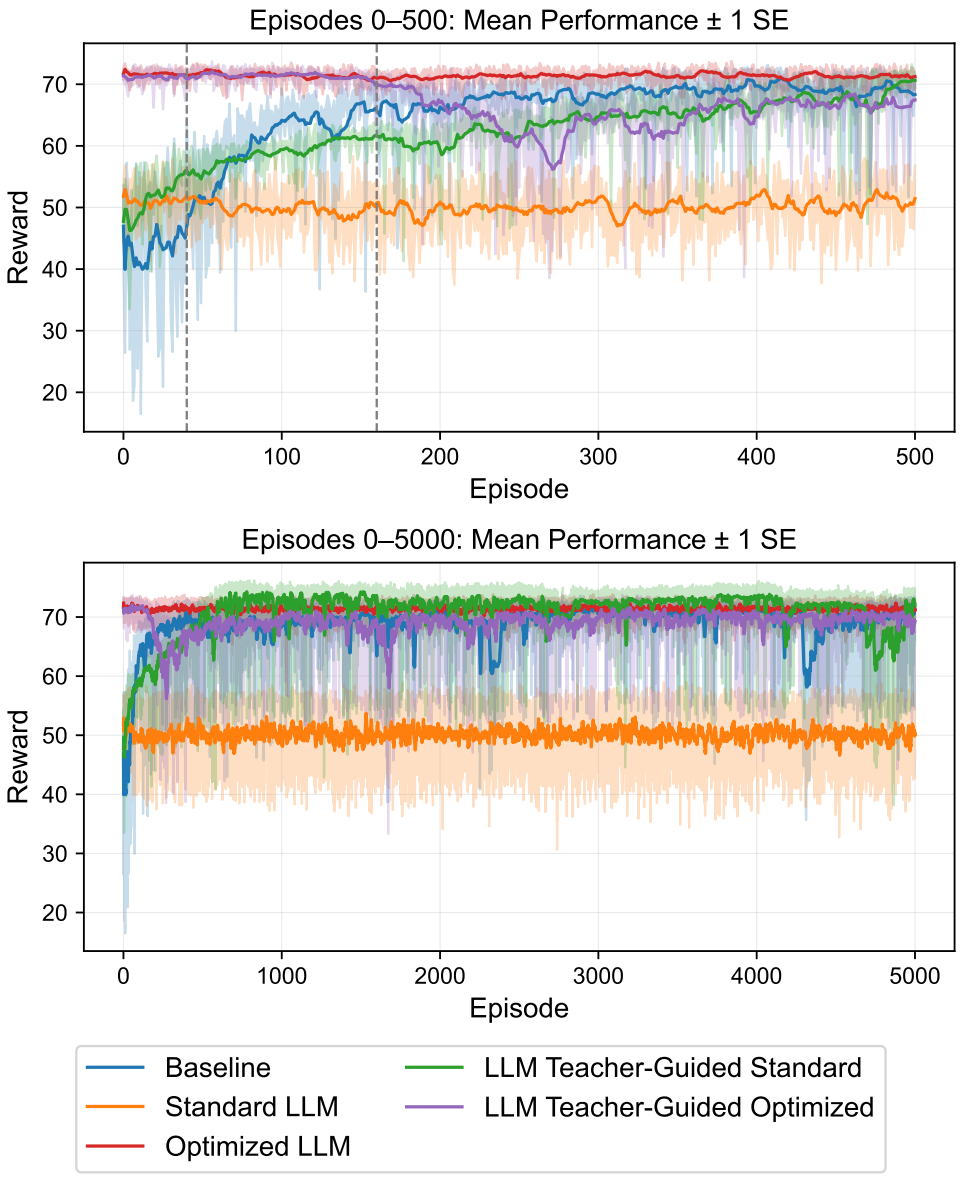


Figure F.3: LLM integration results in the 6-host environment.

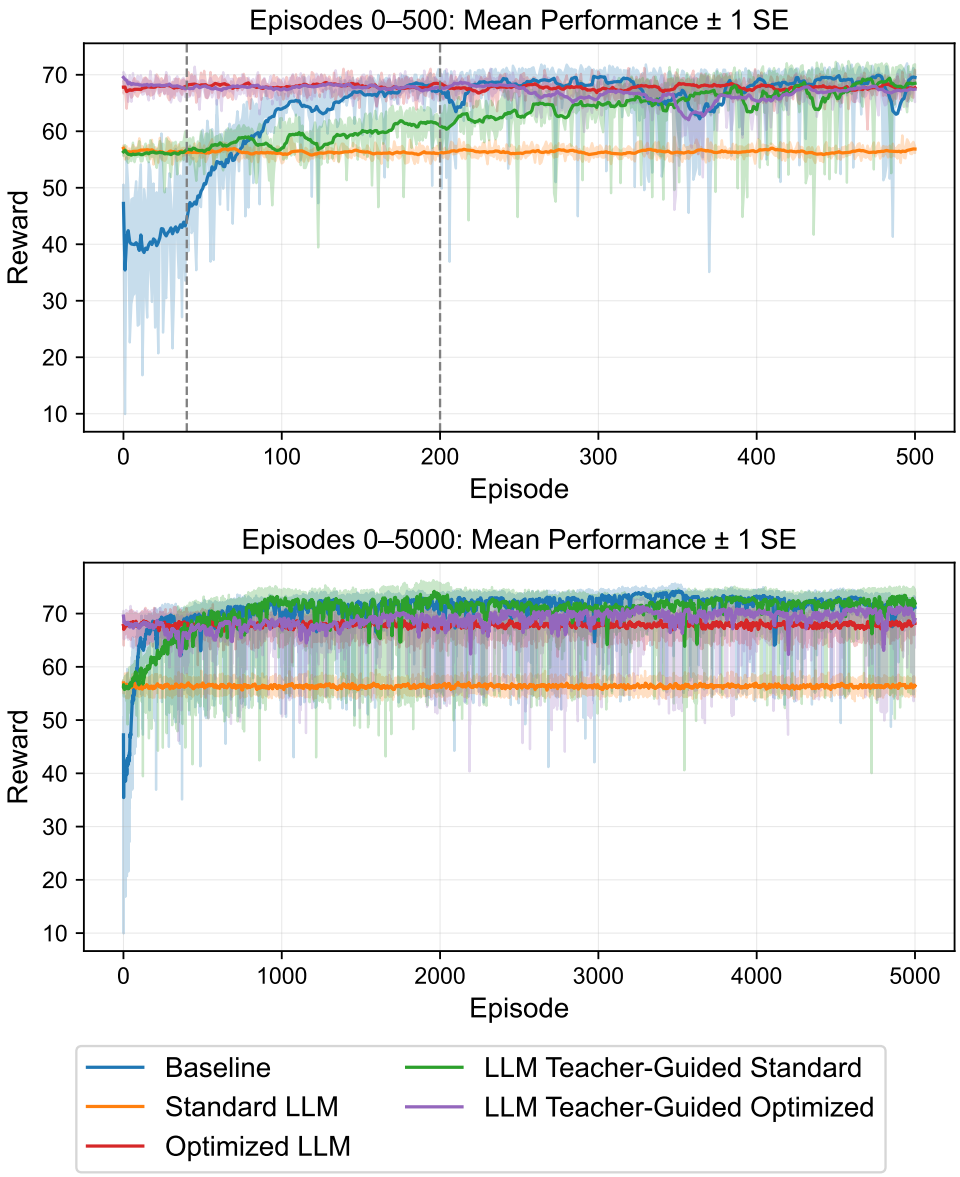


Figure F.4: LLM integration results in the 7-host environment..

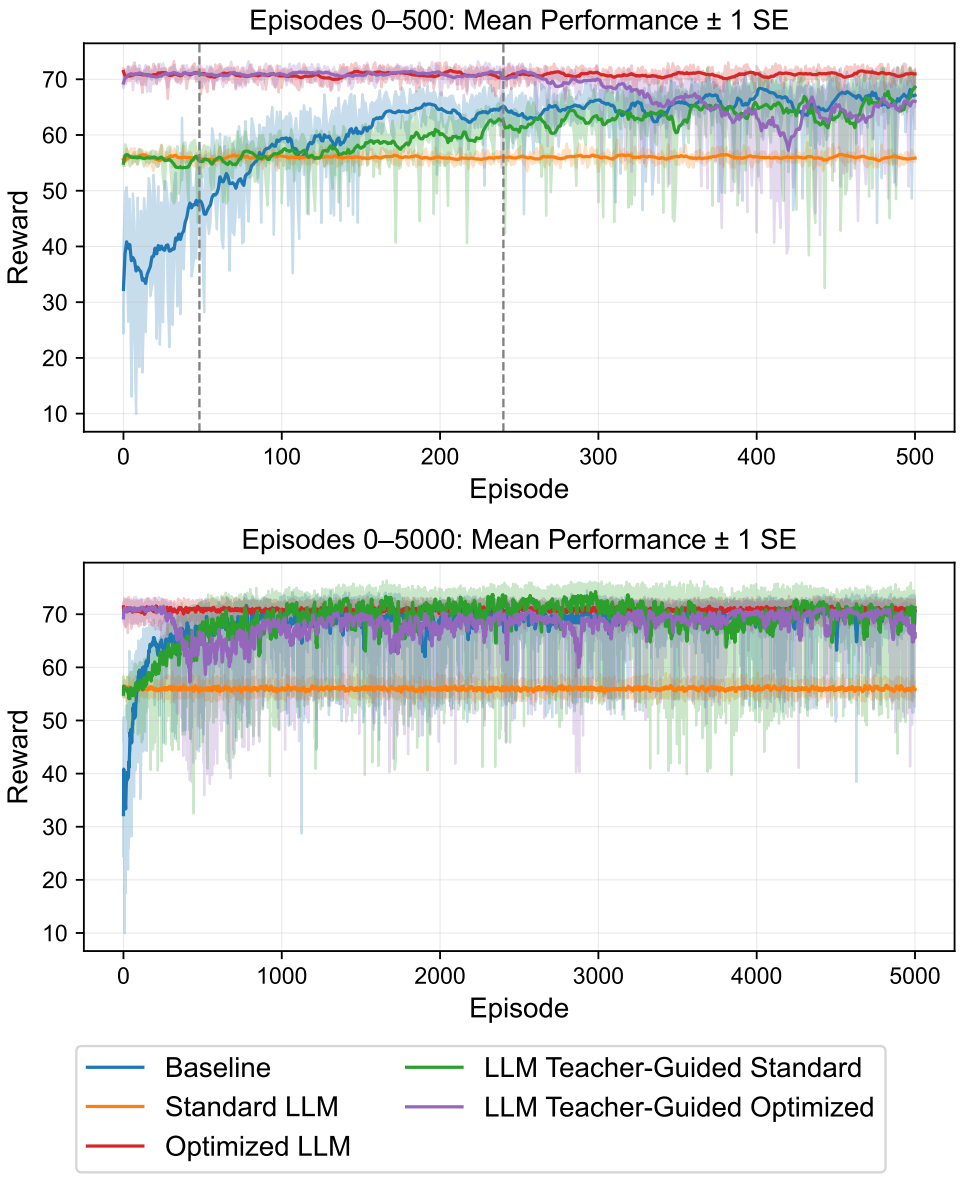


Figure F.5: LLM integration results in the 8-host environment.

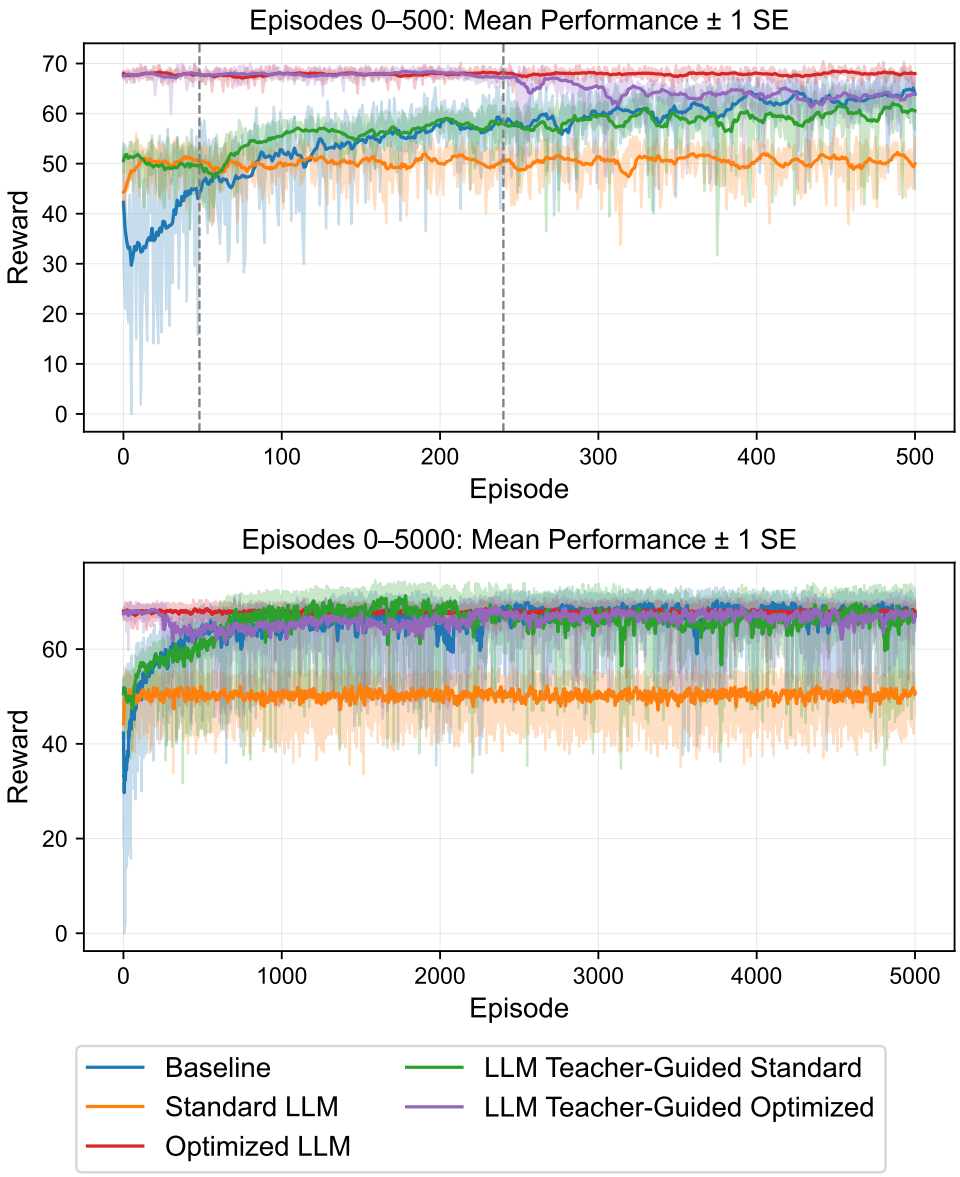


Figure F.6: LLM integration results in the 9-host environment.

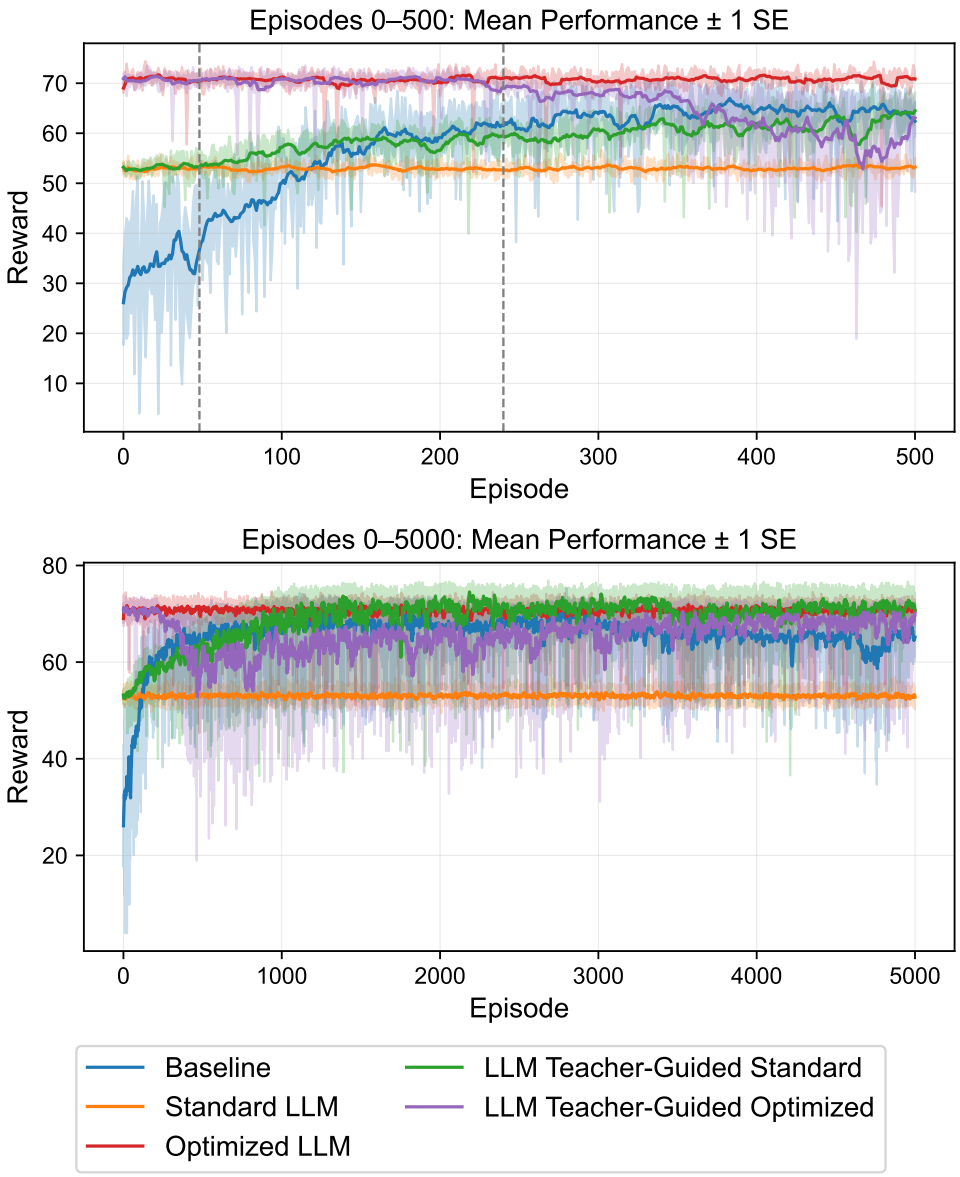


Figure F.7: LLM integration results in the 10-host environment.

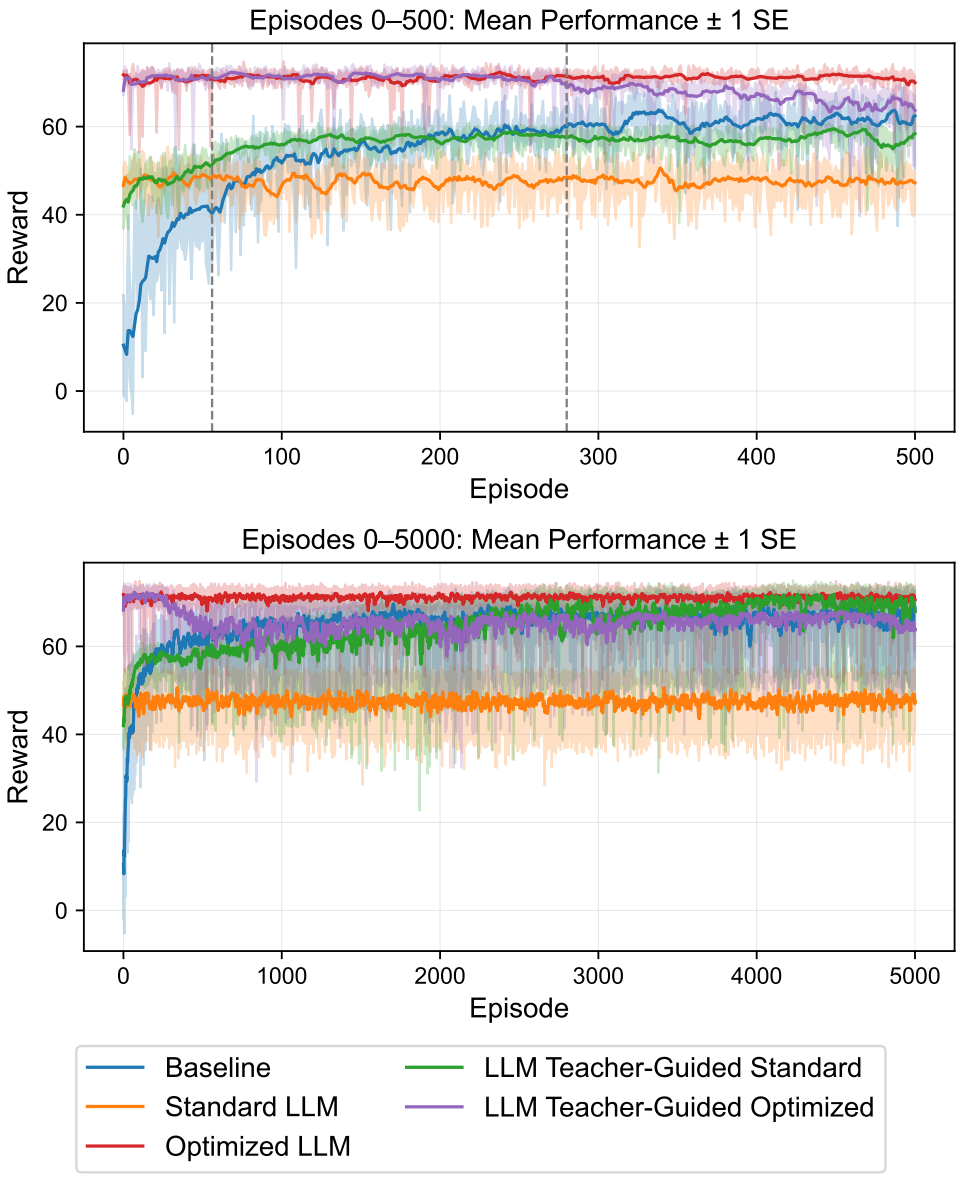


Figure F.8: LLM integration results in the 11-host environment.

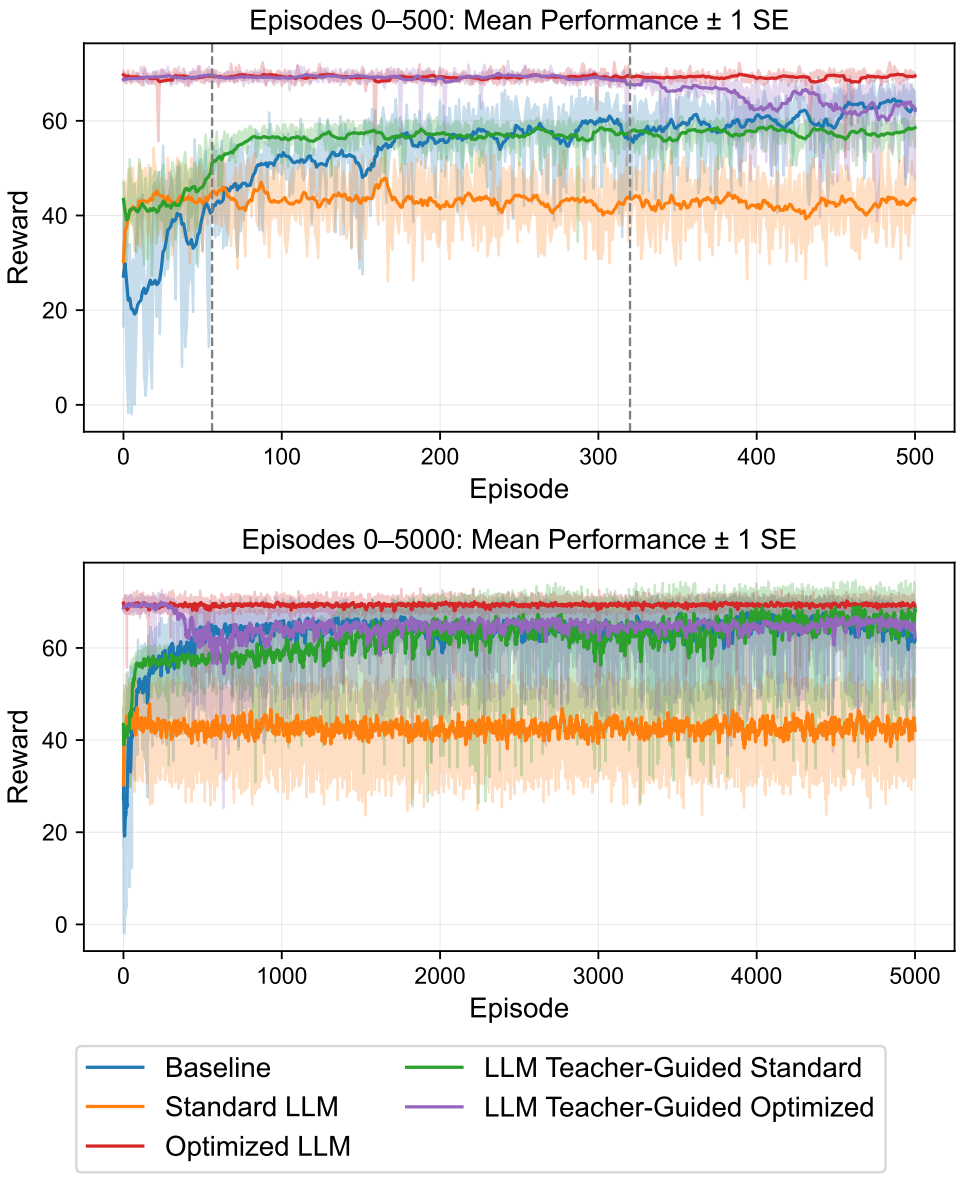


Figure F.9: LLM integration results in the 12-host environment.