AN LLM-DRIVEN FRAMEWORK FOR AUTOMATIC CURRICULUM LEARNING TO ENHANCE GENERALIZATION IN OPEN-ENDED REINFORCEMENT LEARNING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS OF
THE MIDDLE EAST TECHNICAL UNIVERSITY
BY

OLCA ORAKCI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF MODELING AND SIMULATION

JANUARY 2026

AN LLM-DRIVEN FRAMEWORK FOR AUTOMATIC CURRICULUM LEARNING TO ENHANCE GENERALIZATION IN OPEN-ENDED REINFORCEMENT LEARNING

submitted by **OLCA ORAKCI** in partial fulfillment of the requirements for the degree of **Master of Science in Modeling and Simulation Department, Middle East Technical University** by,

Date:

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Olca Orakcı

Signature :

ABSTRACT

AN LLM-DRIVEN FRAMEWORK FOR AUTOMATIC CURRICULUM LEARNING TO ENHANCE GENERALIZATION IN OPEN-ENDED REINFORCEMENT LEARNING

Orakcı, Olca
M.S., Department of Modeling and Simulation
Supervisor: Assoc. Prof. Dr. Elif Sürer

January 2026, 90 pages

Reinforcement Learning research has long sought to achieve generalization. The benchmark environments gradually evolved over the years to achieve this demanding task. As environments became more complex and generalization-oriented, this introduced new research fields, such as Open-Ended Learning (OEL). OEL is the study of learning settings in which a large set of skills exists, some of which may or may not be required for each situation represented in the environment. These complex environments necessitated policies that did not memorize environments but learned generalized skills. With the development of methods like Automated Curriculum Learning (ACL), this challenging task became more achievable. The focus of this thesis is to introduce a standardized way to use Large Language Models (LLMs) as an ACL method, called Adaptive Reasoning Curriculum (ARC), in a novel, high-performing, sample-efficient, and reproducible manner, and to demonstrate the method's effectiveness in a state-of-the-art massively multi-agent OEL environment, Neural MMO 2. The results of the experiments show that the ARC framework outperforms both an expert curriculum and several ACL methods across average return and sample efficiency. With auxiliary helper tools, a dashboard, and a validation tool, ARC framework aims to make OEL research with LLMs available to all researchers and enthusiasts through open source, standardized, and shareable adapters, orchestrators, and experiment files.

Keywords: Generalization, Reinforcement Learning, Open-Ended Learning, Automated Curriculum Learning, Large Language Models

ÖZ

AÇIK UÇLU PEKİŞTİRMELİ ÖĞRENMEDE GENELLEME YETENEĞİNİ ARTIRMAYA YÖNELİK, BÜYÜK DİL MODELİ TABANLI OTOMATİK MÜFREDAT ÖĞRENİMİ ÇERÇEVESİ

Orakcı, Olca

Yüksek Lisans, Modelleme ve Simülasyon Bölümü

Tez Yöneticisi: Doç. Dr. Elif Sürer

Ocak 2026, 90 sayfa

Pekiştirmeli Öğrenme araştırmaları, uzun zamandır genelleştirmeyi elde etme arayışındadır. Kıyaslama ortamları, bu zorlu görevi başarmak adına yıllar içinde kademeli olarak evrimleşmiştir. Ortamların daha karmaşık ve genelleştirme odaklı hale gelmesi, Açık Uçlu Öğrenme (OEL) gibi yeni araştırma alanlarını ortaya çıkarmıştır. OEL, ortamda sunulan her bir durum için gerekli olabilecek ya da olmayabilecek geniş bir beceri setinin bulunduğu öğrenme kurgularının incelenmesidir. Bu karmaşık ortamlar, ortamı ezberlemek yerine genelleştirilmiş beceriler öğrenebilen politikaları zorunlu kılmıştır. Otomatik Müfredat Öğrenimi (ACL) gibi yöntemlerin geliştirilmesiyle birlikte, bu zorlu görev daha erişilebilir hale gelmiştir. Bu tezin odak noktası, Büyük Dil Modellerinin (LLM) bir ACL yöntemi olarak kullanımına dair Uyarlanabilir Akıl Yürütme Müfredatı (ARC) adında standartlaştırılmış bir yaklaşımı; özgün, yüksek performanslı, örneklem verimli ve tekrarlanabilir bir şekilde sunmak ve bu yöntemi en gelişmiş devasa çok-ajanlı OEL ortamı olan Neural MMO 2 üzerinde doğrulamaktır. Deney sonuçları; ARC çerçevesinin, hem ortalama getiri hem de örneklem verimliliği açısından, uzmanlar tarafından hazırlanan müfredatı ve çeşitli ACL yöntemlerini geride bıraktığını göstermektedir. Yardımcı araçlar, bir gösterge paneli ve bir doğrulama aracı, ile desteklenen ARC çerçevesi; açık kaynaklı, standartlaştırılmış ve paylaşılabilir adaptörler, orkestratörler ve deney dosyaları aracılığıyla, LLM destekli OEL araştırmalarını tüm araştırmacılar ve meraklılar için erişilebilir kılmayı amaçlamaktadır.

Anahtar Kelimeler: Genelleştirme, Pekiştirmeli Öğrenme, Açık Uçlu Öğrenme, Otomatik Müfredat Öğrenimi, Büyük Dil Modelleri



To my family

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor, Assoc. Prof. Elif Sürer, for her invaluable mentorship. Her support and insightful guidance throughout my time in the department have been instrumental in my academic growth, encouraging me to exceed my own expectations. I am grateful for the opportunity to learn from her.

Special thanks to my loving wife, Elif Ece Orakcı. Her unwavering support and patience throughout the duration of this thesis, and in the years prior, have been invaluable to me.

My sincere thanks go to my family and friends for their love and support. Their kindness has always been a source of strength, and their help was essential in completing this thesis.

I extend my gratitude to my examining committee members for their time, their willingness to serve on this committee, and their constructive comments.

TABLE OF CONTENTS

ABSTRACT iv

ÖZ vi

DEDICATION viii

ACKNOWLEDGMENTS ix

TABLE OF CONTENTS x

LIST OF TABLES xiii

LIST OF FIGURES xiv

LIST OF ABBREVIATIONS xvi

CHAPTERS

1 INTRODUCTION 1

1.1 Problem Statement and Motivation 1

1.2 Research Questions 2

1.3 Contributions of the Study 2

1.4 Organization of the Thesis 3

2 RELATED WORK 5

2.1 The Evolution of Reinforcement Learning Environments 5

2.1.1 Single-Task Mastery Environments 5

2.1.2 Procedural Content Generation Environments 6
2.1.3 Open-Ended Learning Environments 7
2.2 Curriculum Learning and Automatic Curriculum Learning 9
2.2.1 Domain Randomization vs. Structured Curriculum 10
2.2.2 Intrinsic Motivation and Teacher-Student Frameworks 10
2.2.3 Unsupervised Environment Design and Level Replay 11
2.2.4 Large Language Models as Curriculum Designers 12
3 METHODOLOGY AND EXPERIMENTAL SETUP 15
3.1 Adaptive Reasoning Curriculum Framework 15
3.1.1 System Architecture and Information Flow 16
3.1.2 The Reasoning Model Structure 17
3.1.3 Task Space Management and Curriculum Generation 19
3.1.4 Training Dashboard 21
3.1.5 Validation Tool 25
3.2 Experiment Testbed: Neural MMO 2.0 30
3.2.1 Environment Overview 30
3.2.2 The Flexible Task System 31
3.3 Evaluation Protocol 31
4 RESULTS 37
4.1 Empirical Performance Evaluation 37
4.2 Cost, Speed and Scalability 41
4.3 Qualitative Analysis of Generated Curricula 44

5 DISCUSSION 49

6 CONCLUSION AND FUTURE WORK 53

REFERENCES 55

A ARC FRAMEWORK DETAILS 63

A.1 Agent Details 63

A.1.1 Prompt Stacks 63

A.1.2 Output Formats 64

A.1.3 LLM Call Examples 65

A.1.3.1 Bootcamp Designer 65

A.1.3.2 Training Diagnostician 69

A.1.3.3 Curriculum Architect 72

A.1.3.4 Task Fixer 82

A.1.3.5 Training Summarizer 84

A.2 Validator Tool Reference Tabs 86

B FORMATTING AND NOTATION 89

B.1 General Formatting and Notation 89

B.2 Training and Evaluation Graphs 89

LIST OF TABLES

Table 1 ARC Adapter standard REST API endpoints, covering a large set of needs of the orchestrator and dashboard 17

Table 2 Baseline PPO policy network architecture, encoders and hyperparameters . . . 32

Table 3 Hidden evaluation curriculum task categories predicates and their parameters 33

Table 4 Training, environment and training policy parameters for each run 34

Table 5 System hardware specifications of the experiment device 35

Table 6 Final hidden evaluation task set return for each method, averaged in the last 5M for stability 38

Table 7 Steps to reach baseline maximum performance for each method, shorter step count means higher sample efficiency. 41

Table 8 Mean token usage and cost by Agent Type and LLM Model 41

Table 9 Total cost comparison of LLM models across different optimization profiles . 42

Table 10 Average training duration per method and their duration percentage vs Baseline 43

Table 11 Total LLM calls and duration statistics by agent type and model used 43

Table 12 Average curriculum generation loop duration compared against different optimization profiles 44

Table 13 Task distribution of generated by ARC Framework vs hidden evaluation tasks by predicate type 47

Table 14 Experimental Values and Definitions for values given in Equations 1, 2, 3 . . . 51

Table 15 Notation, abbreviations, and formatting conventions used in graphs and tables 89

Table 16 Smoothing and alignment configurations for line graphs 90

LIST OF FIGURES

Figure 1 Architecture of Adaptive Reasoning Curriculum Framework, showing main components, external services and data flow between modules 16

Figure 2 ARC Reasoning Model flowchart, shows the main method on a high level 18

Figure 3 Curriculum Architect prompt stack, dashed lines represent static content while solid thick lines represent generated content 20

Figure 4 ARC Dashboard main tab, displays general training status, current task details and checkpoints so far to give a high level of understanding. 22

Figure 5 ARC Dashboard Task History Tab, displays statistical data about tasks, their historical change in both high and low level of detail. 23

Figure 6 ARC Dashboard performance tab, displays training metrics as line graphs, with and without TWMEA smoothing. 23

Figure 7 ARC Dashboard LLM decisions tab, displays high level LLM call history, and low level breakdown of all agent calls 24

Figure 8 ARC Dashboard LLM Decisions Tab: Detailed breakdown of agent calls 24

Figure 9 ARC Dashboard LLM decisions tab agent output, displays low level call details, formatted output and full JSON response. 25

Figure 10 ARC Dashboard configuration tab, displays the orchestrator parameters such as model used or checkpoint interval, and verbosity settings. 25

Figure 11 ARC Validator main tab, before uploading validation files, displays file upload panel, quick cheat sheets on the structure. 26

Figure 12 ARC Validator main tab, after uploading validation files, displays completion rate, validated file count, total errors and error breakdown. 26

Figure 13 ARC Validator adapter tab, displays completion of adapter module, endpoint coverage of the adapter, and implementation guide for missing or faulty endpoints. 27

Figure 14 ARC Validator orchestrator tab, displays completion of orchestrator module, agent implementation errors and essential orchestrator function error and their implementation guides 28

Figure 15 ARC Validator config tab, displays orchestrator configurations including LLM parameters, adapter network location and verbosity settings 29

Figure 16 ARC Validator reference documentation, displays subtabs containing high level information of implementation details for each tool. 29

Figure 17 Average return per step on hidden evaluation curriculum for each method 37

Figure 18 Average return per step for each method in their respective training curricula. 39

Figure 19 Explained variance of the policy per step for each method 40

Figure 20 Evolution of curriculum selected by ARC Framework per training checkpoint, visualized using t-SNE Projection 45

Figure 21 All tasks generated by ARC Framework in a one full training, compared to hidden evaluation tasks using t-SNE projection 46

Figure 22 Prompt Stacks for other agents in the ARC Framework, only bootcamp designer is fully static, while other agents use dynamic inputs. 63

Figure 23 ARC Validator reference documentation, adapter API implementation details 86

Figure 24 ARC Validator reference documentation, configuration file implementation details 87

LIST OF ABBREVIATIONS

A3C	Asynchronous Advantage Actor-Critic
ACCEL	Aligned Curriculum through Curation and Evolution
ACL	Automated Curriculum Learning
ALE	Arcade Learning Environment
ARC	Adaptive Reasoning Curriculum
Wandb	Backpropagation Through Time
CL	Curriculum Learning
CoT	Chain-of-Thought
CPU	Central Processing Unit
DDPG	Deep Deterministic Policy Gradient
DQN	Deep Q-Network
DrAC	Data Regularized Actor-Critic
DR	Domain Randomization
ELLM	Exploring with LLMs
GAE	Generalized Advantage Estimation
GPU	Graphic Processing Unit
IDAAC	Invariant Decoupled Advantage Actor-Critic
IMPALA	Importance Weighted Actor-Learner Architecture
LLM	Large Language Model
LSTM	Long Short-Term Memory
MAESTRO	Multi-Agent Environment Design Strategist for Open-Ended Learning
ML	Machine Learning

MMORPG	Massive Multiplayer Online Role-Playing Games
NLE	NetHack Learning Environment
NMMO	Neural MMO
NPC	Non-Playable Character
OEL	Open-Ended Learning
PAIRED	Protagonist Antagonist Induced Regret Environment Design
PCG	Procedural Content Generation
PLR	Prioritized Level Replay
PPG	Phasic Policy Gradient
PPO	Proximal Policy Optimization
PvE	Player-versus-Environment
PvP	Player-versus-Player
RL	Reinforcement Learning
RND	Random Network Distillation
SAC	Soft Actor-Critic
TWMEA	Time-Weighted Moving Exponential Average
TD3	Twin Delayed DDPG
TD	Temporal Difference
TPU	Tensor Processing Unit
UED	Unsupervised Environment Design
VF	Value Function
VPT	Video PreTraining
Wandb	Weights & Biases

CHAPTER 1

INTRODUCTION

1.1 Problem Statement and Motivation

Achieving generalization and robust policies remains a fundamental challenge in Reinforcement Learning (RL). Although most tasks, even complex and specialized ones, can be solved by policies, they struggle to transfer this experience to changes in the environment or the environmental goal. This issue, known as overfitting, particularly affects larger-scale, many-task open-ended environments designed to push the boundaries of generalization and open-ended RL research.

Environments such as Neural MMO (NMMO) 2.0 are ideal domains for this research field. With its massive scale, multi-agent structure, procedural map generation, and rich game mechanics, including survival, combat, and trading, it is a one-of-a-kind environment. This complexity necessitates the use of Curriculum Learning (CL), as without it, the game mechanics are too complex for traditional RL policies to learn. However, designing a curriculum that effectively teaches complex game mechanics that are not necessarily bound together, such as opening a door, requires picking up a key, is notoriously difficult, since there is no optimal curriculum for an environment without a clear goal. In addition, most experts designed curricula assume a strategy is best regardless of the policy state, making these curricula biased and sometimes not even better than randomly selecting tasks.

There are Automated Curriculum Learning (ACL) algorithms that use reward or task signals and policy training metrics to address agents' weaknesses, but these often fail to solve large environments that require complex strategies for long-horizon goals that are only weakly tied. There are some Large Language Model (LLM)-guided curriculum generation methods available, but these are often manually prompted, not scalable, volatile in performance, and challenging to apply across different domains. These limitations are why we created the Adaptive Reasoning Curriculum (ARC). ARC proposes to bridge this gap with an LLM-powered, sophisticated multi-agent reasoning model that adaptively generates a curriculum based on rich, complex signals to theorize about the policy's weaknesses, the strategies to focus on, and the skills to master. This adaptive reasoning method, coupled with a multi-task, multi-agent environment such as NMMO 2, is essential for progress in generalization and open-ended RL.

ARC Framework also aims to make ACL research through LLMs achievable through a standardized decoupled structure where parts of the system can be reused for different en-

vironments. To achieve this we also publish a set of open-source tools for creation, validation and monitoring of the structure. We hope to motivate researchers to share different adapters, orchestrators and configs to achieve better generalization.

1.2 Research Questions

The main research question is whether the ARC Framework can produce high-quality curricula that result in superior generalization and robustness compared to traditional CL and ACL algorithms. This question will determine whether a tailored curriculum that adapts based on policy mastery is sufficient to address the generalization gap in OEL environments. While the quality of the generated curriculum will heavily rely on the quality of the prompt system, we will aim to see if the ARC Framework can detect and target policy weakness, and handle complex prerequisites of the connected skills to achieve meaningful targeting.

Next, we investigate the practicality of the system. Since the main goal of all ACL algorithms is to keep the policy in a zone of continuous learning, meaning the agent is always pushed to learn more difficult configurations, sample efficiency might be on par with final performance. Hence, we will determine whether the ARC Framework is more sample-efficient than traditional ACL algorithms or an expert curriculum.

Another crucial question concerns the system's usability. Since this method uses LLMs to generate curriculum, it accumulates a cost. As the motivation for this thesis is to create a solution available to all researchers rather than a select few, this cost should be small and worth the investment in terms of performance and sample efficiency. We will investigate whether the method is cost-efficient and whether there are opportunities to reduce total costs. Moreover, using LLMs introduces the problem of inference time, which is typically longer than with other ACL methods. We aim to find solutions to utilize LLM-guided, high-quality curriculum generation without drastically increasing training duration.

Finally, we investigate the adaptability of the ARC Framework. Since an important part of our goal is to create a blueprint for the LLM-Guided ACL method, other researchers can easily apply it to other benchmark environments or their custom work. To achieve that, we designed the framework in a modular way, with each component decoupled and able to be used in another setting with minimal change. We also created two additional helper tools that allow monitoring and validation of the components.

1.3 Contributions of the Study

This study has several significant contributions to RL, generalization, and OEL research. First, we propose a novel, highly scalable, customizable, and transferable ACL framework specifically engineered for complex, open-ended environments such as NMMO 2. Unlike traditional ACL methods, which are tightly coupled to the training instance, limited by specific training inputs or by rigid rule sets, this framework is designed to handle stochasticity and the massive scale of OEL benchmarks. One of the most important limitations of these highly dynamic

OEL environments is that they lack native, intrinsic goals, which prevents traditional ACL methods from defining progress and leads to poor performance. ARC proposes a method for defining progress using reasoning models to train robust agents in the most complex virtual worlds.

Another important contribution is the novel ACL method, using flexible training telemetry data, whether it is policy training metrics, evaluation metrics, or additional environment statistical data, as progress indicators. Using an LLM-powered multi-agent reasoning model, ARC Framework can create complex hierarchical connections between environment strategies and go beyond relatively simpler intrinsic metrics or reward signals. This skill enables success in very complex, dynamic environments where rewards are scarce. While these environments are often a nightmare for a traditional ACL method, the ARC framework can use semantic reasoning to compare data across a range of signals from different tasks to produce high-quality analysis and curriculum.

Furthermore, unlike traditional ACL methods that need to feed directly from the training environment and the policy training instance, the ARC framework presents a decoupled structure, separating the system into two core pieces: adapter and orchestrator. This modular design enables two main use cases. First, is the reusability of the components: by changing the adapter slightly for an environment, the user can keep the orchestrator, given that its prompts are not completely fine-tuned for the environment, or modify it very slightly for adaptation. Second, the user can run the adapter and the orchestrator on separate systems or even networks, allowing for the utilization of more system resources for each, which can especially be valuable if the orchestrator is using a local LLM.

Finally, by developing two additional helper tools, one is to monitor the training process, curriculum generation, and llm calls, the other is to validate the implementation of the adapter, orchestrator, and config file, and to publish the framework as an open-source resource, aiming to accelerate OEL research and motivate the RL community to focus on the generalization problem. The structure is designed in a standardized way that can be easily shared and implemented in other environments. Prompts, settings, and the entire run data can be easily exported, reported, or shared. Making it a perfect starting point for a community-driven framework.

1.4 Organization of the Thesis

This thesis consists of six main chapters as follows:

- Chapter 1 is the Introduction. It contains the problem statement and the motivation, the main research questions, and the contributions of this thesis to the field.
- Chapter 2 is the Related Work. It provides background on related topics. It explains the history of RL environments, benchmarks, and some of the most significant RL algorithms developed through them. It also explains CL and ACL in detail, the historical evolution of methods, and the need for a better method.

- Chapter 3 is the Methodology and Experimental Setup. It starts by explaining the limitations of traditional ACL methods and the problems with past LLM-guided curriculum-generation methods, which motivate us to create the ARC Framework. Then it explains the multi-agentic architecture, the methods used to address shortcomings of older methods, and the prompting methods used in the ARC Reasoning Model. It further explains the method's decoupled structure, enabling its flexibility and adaptability. It also explains the two helper tools in detail. Finally, describes the environment used, the experimental setup, the evaluation method, and the baselines.
- Chapter 4 is the Results. It presents the results of the experiments and a qualitative analysis of the generated curricula, covering performance, sample efficiency, cost, speed, and scalability. It also visualizes the created curricula for a quantitative analysis against the hidden evaluation task set.
- Chapter 5 is the Discussion. It distills the information from the results section. Analyzes and compares these results with our goals for this thesis and discusses some interesting findings that can lead to new research.
- Chapter 5 is the Summary. It contains summaries of the section results and discussion, mentions limitations, and includes possible future work.

CHAPTER 2

RELATED WORK

2.1 The Evolution of Reinforcement Learning Environments

RL research trajectory is closely intertwined with the complexity and challenges available in benchmark environments. Historically, these environments have been shifting away from specialization towards more complex domains that demand generalization and open-ended learning. This shift reflects a community-wide recognition that when policies are confined to specific tasks, they inevitably fail when the task definition or the surrounding environment changes or scale [1, 2]. Today, the literature is moving towards more complex environments, where open-ended learning is one of the most researched areas [3].

2.1.1 Single-Task Mastery Environments

Early breakthroughs in RL drew primarily from single-task mastery environments, which served as a foundational benchmark for many research papers. One of these environments is the Arcade Learning Environment (ALE) [4], with an extensive collection of 2600 Atari games, and an alternative for image and ram observation, high-dimensional observation space, it served as a relevant testbed for more then a decade for many RL algorithms.

Deep Q-Network (DQN) [5, 6], an algorithm that uses Q-learning mechanics with deep neural networks to directly learn from high-dimensional inputs. Asynchronous Advantage Actor-Critic (A3C) [7], utilizing multiple parallel agents collecting roll-outs in their respective environments asynchronously to achieve a more diverse experience. Proximal Policy Optimization (PPO) [8] clips the update step to ensure each training step is small and not deviating too far from the previous step. Rainbow [9] is a mix of improvements upon the original DQN, such as Double DQN and Prioritized Learning Replay (PLR). Importance Weighted Actor-Learner Architecture (IMPALA) [10], decouples actors from learners to handle the lag between data generation and model updates. Agent57 [11] uses a specialized exploration agent and has a long-term memory mechanism, allowing it to be the first model to surpass human performance on all 57 Atari games in ALE. MuZero [12] is a model-based algorithm that builds an internal model of the environment to predict rewards and values from hidden states. DreamerV2 [13], another model-based algorithm that can predict future sequences, making it highly sample efficient.

Similarly, earlier simulation environments like MuJoCO [14], originally a high-performance physics engine simulating rigid body control containing eleven early environments, and the DeepMind Control Suite [15] led the progress for continuous control and robot learning, built on top of Mujoco with different robot types called domains with different goals like stand, walk, run, swim and more, by reducing complex signals to few relevant parameters towards the goal, allowing the research for famous continuous action space policies.

Some of these continuous action space policies include Soft Actor-Critic (SAC) [16], an offline policy algorithm that primarily maximizes expected reward and entropy, making it efficient for exploration. Deep Deterministic Policy Gradient (DDPG) [17] is a continuous action space version of DQN using the Actor-Critic mechanism. Twin Delayed DDPG (TD3) [18] is an improvement upon DDPG that uses two Q-Networks instead of reducing overestimation bias, making it more stable during training.

However, while these algorithms were successful in their respective environments, which are characterized by minimal to no variation, even the slightest change in the environment or the goal significantly reduced the policies' performance, suggesting serious overfitting and generalization issues [1, 2]. These limitations created a need in the literature for benchmarks and environments focused on variety, complexity, and generalization rather than specialization.

2.1.2 Procedural Content Generation Environments

With the recognition of this overfitting problem, research turned to expanding environments and introducing unseen elements during training, using methods such as Procedural Content Generation (PCG) [19]. PCG is the algorithmic generation of game content, including items, quests, maps, and whole levels. PCG generates an endless number of unseen environments by algorithmically modifying their layout, appearance, and dynamics. This methodology allowed researchers to train and test their policies on environments that they had not seen before. Because each environment is somewhat different from the previous one and contains different levels and elements, these frameworks are considered benchmarks rather than environments.

Coinrun [1] is a historical turning point in RL research for introducing training and test sets, similar to other fields of Machine Learning (ML), and for using generated environments. Traditionally, policies were trained and tested in the same environments, which often led to overfitting. Using PCG, Coinrun was able to provide unseen test environments, isolating training and test sets. However, even with thousands of procedurally generated levels, policies still overfit rather than learning generalized skills. These findings led to a search for more robust policies that traditionally helped mitigate overfitting in classical ML, such as Data Regularized Actor-Critic (DrAC) [20], a method that uses data augmentation, such as cropping and rotation of observations, to regularize the policy and improve generalization. Another method is Invariant Decoupled Advantage Actor-Critic (IDAAC) [21], which decouples the policy and value functions to reduce the effect of visual noise on the perceived task, helping mitigate overfitting common in generalization.

Procgen [22] took many lessons from the Coinrun benchmark to set a new peak, and quickly became the standard benchmark for generalization in RL. With 16 arcade-style environments

(which also included Coinrun), a unified 64x64x3 image-based observation space, an API standardization similar to ALE, and with procedurally generated parameters for variance, it became a testbed for popular algorithms like Phasic Policy Gradient (PPG) [23], which showed decoupling policy and value functions significantly increased sample efficiency in high-variance environments. This algorithm also inspired the IDAAC, which added an auxiliary loss mechanism to encourage policy representations to focus only on task-relevant features. Apart from policy algorithms, Procgen is also used as a testbed for unsupervised environment design algorithms, such as Prioritized Level Replay (PLR) [24], which serve as curricula to maximize policy training efficiency.

Avalon [25] addressed a crucial complexity weakness in other benchmarks, which were mainly focusing on arcade-like games. With a high-fidelity 3D survival environment, it introduced physics-based interactions, tool use, and navigation across procedurally generated topographies. It quickly became a key benchmark in Offline RL Generalization research [26], showing that agents trained on offline data struggled significantly more than online agents when transferred to an unseen environment. Showing the fragility of offline datasets in highly complex and dynamically changing environments.

Griddly [2] introduced a highly optimized, high-speed, configuration-based generation grid-world-type benchmark. While grid-world benchmarks seem outdated compared to other benchmarks, their ability to calculate millions of steps per second made them highly popular in computationally expensive fields such as early Open-Ended Learning and Multi-Agent RL. It served as a testbed for the Multi-Agent Environment Design Strategist for Open-Ended Learning (MAESTRO) [27], a multi-agent curriculum-learning algorithm that creates zero-sum curricula to train robust policies against varying opponent strategies.

MiniGrid & MiniWorld [28] is a minimalist, modular, and goal-oriented grid-world benchmark. While it is excellent for rapid prototyping, its significance lies in BabyAI [29], a framework built on top of the Minigrid benchmark with natural language instructions, making it an excellent benchmark for Grounded Language Learning research. Also, due to Minigrid's highly sparse reward structure, it serves as a key benchmark for intrinsic motivation and exploration algorithms. RIDE [30] incentivizes agents to choose actions that cause significant changes in learned state representations, which causes the policy to interact with objects rather than skip them while chasing reward accumulation. BeBold [31] focuses on "Noisy TV Problem" [32], an effect where the agent gets obsessed with a random, noisy signal. Since curiosity-driven algorithms urge policies to explore states it cannot predict well, this noise can act like a pit where the agent gets stuck. BeBold implements a novel challenge rule that allocates intrinsic rewards only to states that are novel relative to their history. Both of these algorithms focus on sample-efficient exploration.

2.1.3 Open-Ended Learning Environments

While PCG addressed some of the roadblocks on the path to generalization, long-horizon complexity and massive-scale tasks remained unaddressed. These tasks, which are often difficult for humans alike, require more advanced methods rather than increasing the training set. These limitations led the RL community to another hot research area in the domain:

Open-Ended Learning (OEL) [3, 33]. These environments require policies that are robust to changes and non-stationarity, possess complex skill-learning capacity, and support diverse interactions [34, 35]. They are also often testbeds for research on sample efficiency and CL.

MineRL [36] uses the sandbox nature of Minecraft for long-horizon planning and as an imitation-learning benchmark. Unlike other benchmarks, MineRL encourages human demonstrations to solve hierarchical tasks. While some tasks are more manageable, such as chopping wood, others are difficult even for human players, such as obtaining diamonds. For Minecraft, the best-selling game of all time, finding gameplay footage is straightforward. Utilizing this, OpenAI published Video PreTraining (VPT) [37], a foundational MineRL agent, using 70,000 hours of unlabelled Minecraft gameplay footage using an Inverse Dynamics Model to label actions. The resulting policy demonstrated that, given sufficient data, human-level skill was achievable in OEL environments, which had previously seemed almost impossible with pure RL training.

NetHack Learning Environment (NLE) [34] is one of the most difficult benchmarks in RL. It is a terminal-based roguelike environment with procedurally generated levels. Maybe one of the most significant findings during the NeurIPS 2021 NetHack Challenge [38] was that rule-based agents outperformed state-of-the-art deep RL agents, highlighting how weak current algorithms are for long-horizon and long-term-strategy planning. The authors of the environment and the competition later published a large dataset using the winner of this competition, a symbolic bot that does not use deep RL, to be used as a teacher in distillation methods [39].

Crafter [40] is a 2D open-world survival game that requires solving 22 hierarchical achievements through exploration and long-term reasoning, and is one of the primary benchmarks in Model-Based RL and intrinsic motivation research. The first model to solve this environment, although its results were not human-level, was DreamerV3 [41]. DreamerV3, like its previous version, DreamerV2, is a model-based method that uses discrete latent states and symlog predictions, making it suitable for a wide range of domains. This result showed that, with a model capable of predicting long-term outcomes, it is possible to solve long-horizon games without relying on heuristics.

XLand-MiniGrid [42, 43], inspired by DeepMind's massive-scale Meta RL environment XLand [33], bridges the gap between cutting-edge and accessible benchmarks. Utilizing JAX [44], a high-performance Python numerical computation library engineered for ML research on Graphics Processing Units (GPUs), and Tensor Processing Units (TPUs), for its high speed, and built on top of Minigrid, Xland-Minigrid was able to support training thousands of agents in vast procedurally generated worlds. Due to its speed and ability to handle many agents, it became a primary benchmark for Ad-Hoc Teamwork and zero-shot generalization research, enabling agents to collaborate in unseen environments on the fly.

Neural MMO [45, 46] is a massively multi-agent and highly complex OEL environment, where 128 player-controlled agents and a high number of Non-Playable Characters (NPC) compete against each other, in a procedurally generated world. With the release of Neural MMO 2.0, a flexible task system was introduced, making it the primary benchmark for Curriculum Learning at scale. Since it is a long-horizon game with complex and highly dynamic elements, such as 128 agents and 256 NPCs, most modern policy frameworks were not up to

the task when it came to training in this benchmark, due to massive rollouts.

One of these important RL libraries is CleanRL [47]. CleanRL focuses on code simplicity and understandability. As an informative baseline, it provides single-file implementations of the most popular RL algorithms. PufferLib [48], on the other hand, focuses on implementability and speed. Many new environments, due to their complex and slow mechanisms, are not efficiently utilized by traditional implementations of RL algorithms. Pufferlib solves this by wrapping these new and popular environments and converting them to flat, vectorized formats. This feature enables simplicity, speed, and scalability, allowing the training of more diverse policies in environments that were previously too slow to support it.

2.2 Curriculum Learning and Automatic Curriculum Learning

RL methods showed incredible potential in many fields surpassing human players with ease, such as ALE [6] where DQN famously reached human level on 75% of environments or when AlphaStar [49] beat professional players, and achieved Grandmaster rank in StarCraft II. Despite success in many complex environments and human-level performance across various domains, previously unobserved elements or slight changes to environmental parameters still significantly affect these policies [50, 51, 52]. While introducing PCG indeed increased the variety of the training set and produced more robust policies, it also introduced new problems, such as sample inefficiency and difficulty with exploration.

In early OEL environments, such as NLE or MineRL, using traditional deep RL algorithms failed due to the massive size of the state space, highly sparse rewards, and long-term planning requirements. Since these algorithms often relied on random exploration until stumbling upon a reward to start learning, environments with such sparse, hard-to-get rewards are often a nightmare. That is why in these environments, imitation learning and symbolic bots were much more successful than RL algorithms. To address this problem, the literature turned to Curriculum Learning (CL) [53], a training structure in which agents are trained in an organized, increasingly complex manner, akin to a human student's curriculum.

While CL, in theory, is straightforward and achievable for simple tasks, benchmarks that require it are seldom simple. Unfortunately, creating a curriculum for a high-variability, open-ended environment is challenging, as it requires expert-level knowledge of the benchmark domain and, most likely, a well-tested strategy to beat it. However, most OEL environments are highly dynamic and challenging to plan for, and some are multi-agent, making it very difficult to foresee agents' trends, which led the researchers towards Automatic Curriculum Learning (ACL) [54], where curriculums, whether new tasks or procedurally generated environments, are generated automatically and adapted algorithmically to match the policy level in the fly.

2.2.1 Domain Randomization vs. Structured Curriculum

One of the most well-known approaches in environments with high variation is Domain Randomization (DR) [55]. DR is an approach in which environmental parameters are sampled randomly at each training episode, meaning that, depending on the variable, it could be a random level, difficulty, terrain, goal, or anything else without discrimination. The main goal is to have the policy experience a training set so diverse that the test environment will be another environment similar to what the agent already experienced during training. DR is effective in some cases, such as transfer learning in robotics where complex learning algorithms often overfit to parameters in the simulation, and randomizing these simulation parameters like joint friction, mass or actuator latency better generalization can be achieved [56]. For generalization to to be successfully achieved, the policy needs to master the entire distribution of the parameter space. In high-variance PCG environments or long-horizon tasks, due to the randomized nature of the DR environment, task difficulty can vary widely. Agents are often training on challenging tasks or environments that they cannot currently solve, so they gain no benefit. On the other hand, this could also be the other way around, where sampled parameters produce too easy tasks or environments, yielding no benefit, both cases wasting computational resources and reducing sample efficiency [57]. These inefficiencies became more evident as the benchmarks became more high-variance and long-horizon heavy, showing the need for structured curricula where the agents are always kept in "Zone of Proximal Development" [58], which is a sweetspot between easier and more difficult parameters where agents' current capabilities are not quite there but certainly achievable. Unlike static, always-randomly sampling DR, structured curricula are designed to dynamically adapt to current agent capabilities, aiming to keep agents in this zone.

2.2.2 Intrinsic Motivation and Teacher-Student Frameworks

Creating a curriculum that adapts to current agent capabilities and dynamically adjusts difficulty or focuses on agents' weaknesses is not a simple task. This is where ACL algorithms come into the picture, using a structure that dynamically adjusts parameters based on signals from the training environment. A very popular ACL method is the Teacher-Student Framework [58]. In this context, the "Student" is the RL agent, for example, a PPO agent. The student is trying to maximize extrinsic rewards to solve the tasks. The "Teacher" is a whole different algorithm, such as a generative network. The teacher's objective is to select the optimal sequence of tasks or environments to maximize the student's learning progress. It is often a loop in which the teacher suggests tasks, students attempt them, and report the outcomes to the teacher, who then updates their own strategy based on the students' success and trajectory.

For the teacher, the most challenging part is to differentiate a "good task" without human supervision. Since the parameter space is often huge, the number of possible tasks is almost endless. Intrinsic Motivation (IM) is a key mechanism, inspired by developmental robotics [59]. IM generates intrinsic reward signals, mimicking drives such as curiosity or progress from policy rollouts. These intrinsic rewards are crucial in environments with long-horizon tasks or very sparse reward signals. Teachers can use them to identify which tasks are too easy or too difficult, and students can use them as rewards for exploration.

One of the most robust IM signals is Learning Progress (LP) [60]. It is a signal that indicates where the agent is learning and improving the most. This metric helps identify good tasks: a teacher needs to identify the zone where tasks are neither too difficult nor too easy. Hence, if agents always fail to earn rewards, the progression is zero; if they always earn the maximum reward, the progression is zero, which is known as the theory of maximizing the information gain for the policy [61].

While IM is a great way to find "good" tasks, it is often used directly by the student as well. In long-horizon environments with sparse rewards, these intrinsic reward signals encourage meaningful exploration by rewarding novel states, as in methods such as Random Network Distillation (RND) [62] and Count-Based Exploration [63]. Methods that combine LP for the teacher and IM rewards for the student often perform well in OEL environments [64].

2.2.3 Unsupervised Environment Design and Level Replay

Traditional ACL algorithms select tasks from predefined datasets, but OEL environments have an endless space of possible configurations. This endless parameter space challenges the idea of creating a dataset and having the algorithm select the best possible tasks from it. Since the dataset may be too large to cover every possible combination, it is difficult to select good tasks; if it is too small, the agent cannot explore some important combinations. To address this limitation, Unsupervised Environment Design (UED) was introduced [65]. UED also uses a Teacher-Student based structure, but instead of finding the base task using the IM signals, it generates environments to maximize the students' regret, effectively finding out where the agents are performing poorly, compared to an optimal or better performing agent.

One of the earliest algorithms to use this structure is Protagonist Antagonist Induced Regret Environment Design (PAIRED) [65]. PAIRED uses a novel structure that changes the classical Teacher-Student method. It implements three roles: the protagonist (effectively the student), the antagonist, and the teacher. The antagonist is either an optimal agent or a better-performing model that can solve more difficult tasks compared to the protagonist. The teacher uses this better agent to generate tasks that the antagonist can solve, but that are challenging for the protagonist, ensuring the generated tasks are achievable and that the protagonist is getting new information. While theoretically sound, PAIRED still struggled with sample efficiency and finding the Zone of Proximal Development.

Another very popular ACL algorithm is Prioritized Level Replay (PLR) [24]. PLR does not generate a new environment every time, as other ACL algorithms do; it acts as a curator. PLR samples random levels for the agent and tracks the agent's success. If a level has a high learning potential, high Temporal Difference (TD) error [66], which is a metric used to predict value by looking at successive predictions and their difference, and value loss, it "replays" the level for the agent, since it provides the agent with something to learn. This way, the agent has a level it can learn from, while PLR actively samples other random levels to get more replays.

While PLR is one of the most successful algorithms of its time, some hybrid algorithms

use curation methods alongside evolutionary search rather than random sampling, such as Aligned Curriculum through Curation and Evolution (ACCEL) [67]. ACCEL starts training very similarly to PLR, randomly sampling levels, but when it has some replays, it mutates these levels to explore a more diverse space, without expanding too far into unsolvable territory. This evolutionary method proved to be one of the best for solving sparse-reward environments.

2.2.4 Large Language Models as Curriculum Designers

The recent popularity of Large Language Models (LLMs) also affected the ACL literature. Due to LLMs' reasoning capabilities, they have the potential to replace heuristic or bandit-based Teacher algorithms. Traditional ACL algorithms use signals to understand the features and goals of environments, but LLMs can apply their common sense to understand the mechanics of benchmarks and tasks. This understanding allows them to act as instructional teachers and create complex, strategic curricula, just as experts would curate one.

One of the most well-known advantages of LLM-guided ACLs is that pretrained models are familiar with benchmarks and well-researched agent strategies, and can apply knowledge from similar environments or games to the current task space. One of the LLM guided ACL methods is Exploring with LLMs (ELLM) [68]. ELLM uses LLMs' reasoning capacities to suggest diverse, semantically meaningful tasks for agents in the Crafter environment, efficiently exploring the environment where traditional random network distillation fails. Similarly, Eureka [69] leverages LLM code-generation capabilities to generate dense reward functions in sparse-reward environments, using training signals and human evaluator comments to feed the LLM, which later evolved to only training feedback in their next paper [70]. Eureka showed that LLMs can perform better reward shaping in NVIDIA IsaacGym simulator environments [71], which is a high-performance physics simulator, supporting GPU computation of the environment steps, which used to cause bottlenecks in traditional Central Process Unit (CPU) powered simulators, since transferring CPU-bound data from environment to GPU-bound policies is dreadfully slow.

LLMs can also generate meaningful environments within a well-constrained parameter space. EnvGen [72] creates environments inside the parameter confinements of the Crafter environment, depending on the agent's weakness. By identifying key failures such as "agent fails when obstacles are red", it generates more environments with red obstacles, strongly mimicking a human instructor. The authors of the paper also note that the underlying LLM often generates logically sound environment configurations that are not functionally correct, pointing to weak grounding and hallucination problems, leading the LLM to imagine it is covering the task domain while it is not.

Despite their promise, LLM-guided ACL methods often suffer from critical issues, making it difficult for the community to adopt them. These problems are mostly grounding and hallucination caused by "Reasoning Gaps" [73], where LLMs generate tasks that are often semantically correct and plausible but logically impossible or not possible yet. Since LLMs choose the next word based on the most likely word during pretraining, they require firm grounding and prompt engineering. Furthermore, due to the inherent randomness of the generated content, regardless of the input, their achievements are difficult to reproduce [74].

Another significant issue is their cost: since most configuration spaces are quite large and training sessions are long, API call costs add up, limiting the use of state-of-the-art models by small research groups lacking financial support. An alternative is to use smaller models, but they often lack the capacity and reasoning depth required for complex planning.

CHAPTER 3

METHODOLOGY AND EXPERIMENTAL SETUP

3.1 Adaptive Reasoning Curriculum Framework

The most pressing challenge with these LLM-guided ACL methods is hallucination. Hallucination in the context of LLMs is the generation of content that is either factually incorrect, irrelevant to user input, or contradictory to previously generated content. This issue is a well-researched phenomenon [75], and is the most crucial problem while applying LLMs to real-world tasks. In the context of ACL, hallucination often refers to the generation of incorrect tasks, goals, or reward functions, rendering these outputs either unusable or not beneficial. Since the next generations and the performance of the agents are tied to these false generations, this problem often makes LLM-guided ACL methods unreliable.

Since LLMs are pretrained on vast datasets, they tend to complete a sentence in the most "semantically correct" way. This bias towards certain words after a sequence often makes the curriculum semantically and logically correct but strategically weak. For instance, during a generation of a task that starts like "pick up ..." agent can choose to fill this with semantically correct "water", while this is also logically correct, if there are no strong grounding present, and if a prerequisite like the agent needing a bucket to be able to pickup water exists, than this task while definitely valid becomes strategically weak [76]. This issue, known as the grounding problem in LLMs, requires the model to understand its environment and surroundings to avoid such mistakes when generating tasks. However, it is not a trivial task to ground an LLM for a real-world task.

Another well-known problem with LLMs is the Lost-in-the-Middle phenomenon [77]. Due to LLMs' U-shaped attention span, which focuses on the beginning and end of the context, crucial aspects of the input in the middle are often overlooked during generation. While this shape helps LLMs understand their persona, which is often located at the beginning of the prompt, and the task, located at the end, some crucial details or the task's rules get lost in the context. This problem becomes more significant as the context size increases.

Due to these problems, LLM-guided ACL methods are often specialized for a single environment and lack the means to apply them across different benchmarks. Researchers often tune their prompts to the letter to avoid these issues and to get good and stable results, confining their method to a single environment.

We designed the Adaptive Reasoning Curriculum Framework (ARC) to address common problems in LLM-based ACL methods. With its multi-agentic structure and agents specialized to handle distinct parts of the planning problem, ARC aims to address these issues and make LLM-guided ACL accessible to everyday users with a novel, user-friendly framework.

3.1.1 System Architecture and Information Flow

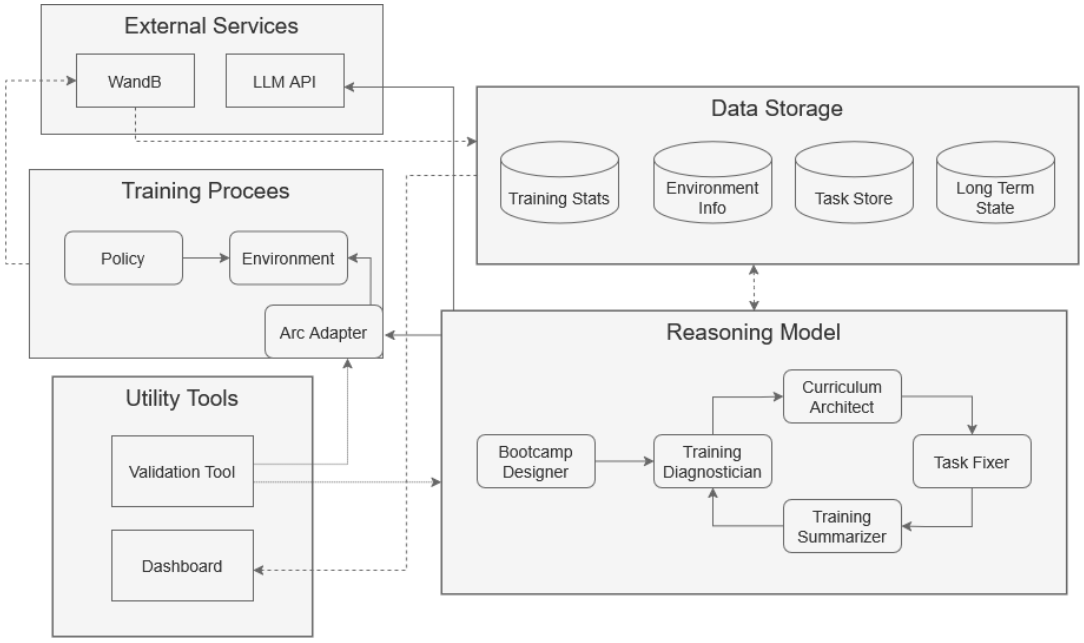


Figure 1: Architecture of Adaptive Reasoning Curriculum Framework, showing main components, external services and data flow between modules

The ARC Framework is designed to prevent known problems in LLM-guided ACL algorithms actively. With a novel information distillation structure and multi-agentic flow, it aims to make the data it processes maintainable. Moreover, with an uncoupled structure, where the special ARC adapter created for the environment tracks the training and injects the generated tasks, and a reasoning model separately distills the information and plans the next curricula. The adapter also standardizes the environment and allows the reasoning model to be applicable between different environments with minimal change.

The ARC adapter is a REST API server that allows seamless interaction between the training instance and the reasoning model. With its standard endpoints, shown in Table 1, it is intended to serve as a bridge that allows the reasoning model to freely manipulate the training process. Since all environments and policy frameworks have their own structures, the ARC adapter should be configured so that it at least controls the start of training, can track progress, and sets the tasks used by the environment. In frameworks that do not log to Weights & Biases (Wandb) [78], an online experiment tracking platform where live training parameters can be logged in real time to create complex analysis graphs and tables later, the adapter should be

set to log important metrics for the reasoning model to track.

Table 1: ARC Adapter standard REST API endpoints, covering a large set of needs of the orchestrator and dashboard

Method	Endpoint	Description
GET	`/`	Root endpoint with usage info
GET	`/status`	Get curriculum status and training metrics
GET	`/get_tasks`	Get all available tasks
GET	`/health`	Health check
POST	`/start_training`	Start training (releases wait lock)
POST	`/complete_training`	Mark training as completed
POST	`/set_task`	Set task(s) with optional weights
POST	`/add_task`	Add a new task
POST	`/batch_add_task`	Add multiple tasks
POST	`/generate_replay`	Generate replay file

The reasoning model consists of five specialized agents. Each focuses on a specific part of the curriculum, allowing better grounding and reducing effects such as context poisoning and the Lost-in-the-Middle phenomenon. The main structure of the agents and their information flow is shown in Figure 1. While one agent is called only once per training session, other agents are called in a loop, with some special conditions that can skip agents.

3.1.2 The Reasoning Model Structure

The reasoning model is designed with two specific goals in mind. The first goal is to address the long-standing hallucination problem that made LLM-guided ACL unstable and challenging to replicate before. The second goal is to create a blueprint that is easy to replicate across other use cases, whether in a new environment or a multi-environment benchmark. It has a five-agent structure, all specializing in particular tasks to reduce the complexity load on the agents. Structure flow is one agent run once, and the other four agents are called in a loop during training, with some basic rules shown in Figure 2 at each checkpoint interval.

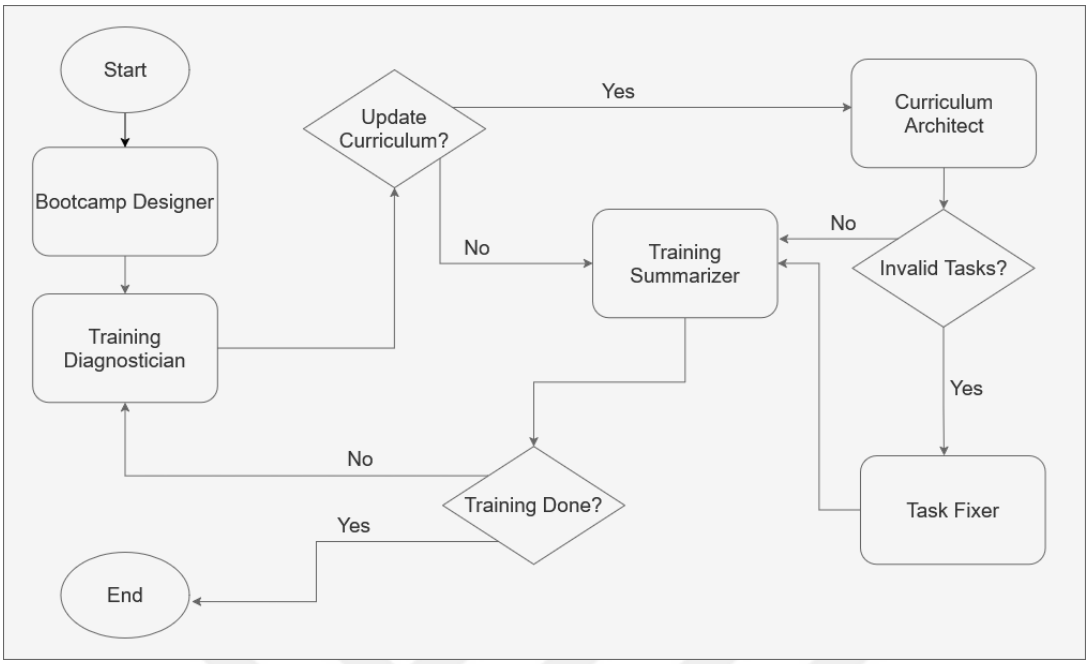


Figure 2: ARC Reasoning Model flowchart, shows the main method on a high level

The first agent in the reasoning model is the Bootcamp Designer. It is the only agent that does not rely on training data, since it is built to create a preliminary curriculum for initial exploration of the environment. It is encouraged to create a diverse task space to gain a high-level understanding of which tasks are more manageable and can be learned early, and which require prerequisites to tackle. In this context, it mainly uses environmental information data and environmental output formatting to generate tasks. Bootcamp Designer is called once per training, while other agents will be enhanced further with training insights. It is recommended to provide clear environment information to prevent grounding issues with this agent.

The second agent is the Training Diagnostician. Training Diagnostician is called at each checkpoint to analyze the current status of the training using training stats, such as current iteration task completion rates, average return, loss metrics, and other environment stats, if available. To avoid Lost-in-the-Middle phenomena, it is better to have a mechanism that converts the raw graphs to a more reasonable format, such as % change or trend. While the output of this agent is used directly as context in other agents, it also serves another crucial task. This task is to decide whether to update the current curricula and, if so, which type of behavior would be best suited: adjusting weaknesses, increasing difficulty, or reducing difficulty. Depending on the update decision, the orchestrator will skip the curriculum-generation-related agents, and if the update decision is given, the update behaviour will also be passed.

To avoid confusion, it would be better to explain the fifth and final agent, Training Summarizer, before the curriculum-generation-related agents. Training Summarizer is the long-term state generation agent. It is tasked with summarizing the analysis, curricula focus, and training strategy for each checkpoint and updating the previous state accordingly. This state helps the curriculum generation strategy be more consistent, less repetitive, and more focused. Summa-

rizing the state rather than passing it as is helps reduce Context Poisoning and keeps generation costs low.

The third agent, and probably the most important one, is the Curriculum Architect. Curriculum Architect is the agent that creates the curriculum the policy trains on, so it must have all the necessary information from both static sources, like environment information, generation rules, output format, and dynamic sources, like Training Diagnostician analysis, Training Summarizer long-term state, and training metrics.

The fourth agent is the Task Fixer. It is a precaution against hallucinated task arguments, since, despite all grounding efforts, the LLM still contains its foundational data, which can include old parameters from earlier versions of the environment. Also, using dynamically generated content as a prompt, even with strong guidelines to ensure grounding, could introduce some noise, such as a different wording for world dynamics. Task Fixer only focuses on task generation notation and has no planning goal; the input provided contains only the task generation rules and parameters. Due to this light context, it can easily interpret Curriculum Architect's intention for failed tasks and fix them.

There are two main faulty task types: the first is tasks returned by the adapter with a fault flag. These tasks have incorrect parameters in the task space, which causes an exception in the adapter task registry. The Task Fixer agent fixes them. The second type of fault is more difficult to account for. Even with the final measure, the Task Fixer agent, it is not always possible to catch hallucinations or functional but still faulty tasks whose syntax is completely valid, so they are registered as tasks. But learning it brings no value, such as trivial tasks where the agent succeeds without any mastery, or impossible tasks like gathering a large amount of gold in a single episode. These tasks are unfortunately trained in cycles, then evaluated by the Training Diagnostician. In most cases, the Training Diagnostician agent marks them as redundant, and they are not repeated during curriculum generation.

3.1.3 Task Space Management and Curriculum Generation

The most crucial part of the ARC Framework, as with any other ACL algorithm, is the generation of the actual curriculum. While all other parts work in harmony, the main goal is to increase the quality of this generation. For proper grounding, the agent should have all the information it needs before generation. The prompt construction order is shown in Figure 3.

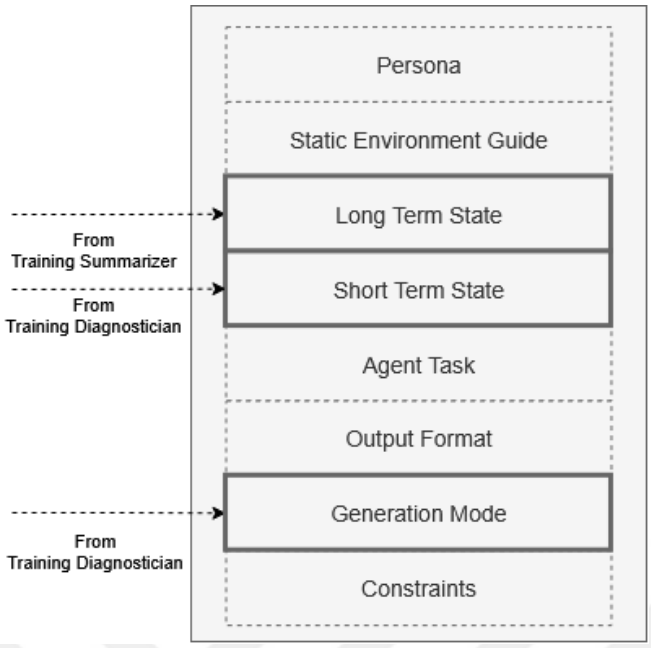


Figure 3: Curriculum Architect prompt stack, dashed lines represent static content while solid thick lines represent generated content

The prompt starts by setting the agent's persona, which sets the tone for the rest of the prompt. Transformer models exhibit a "Attention Sink" phenomenon [79], leading to excessive attention at the beginning of the prompt. Persona is a key element that we cannot afford to be washed out by the rest of the context.

The second part is the Static Environment Guide, which includes environment details, task parameters, and insights into task difficulty. This part requires some knowledge of the environment, even if it is old, and the LLM has plenty of knowledge about it. This requirement is because most environments have versions: some introduce new mechanics, while others remove them, which could result in invalid tasks being generated. Also, the quality of this field might be one of the most important parts of the prompt for grounding the agent, since it essentially confines the agent within the environment's parameters. This importance is why we place it just after the persona, since it is close to the beginning, where we know attention will be.

The Third and Fourth sections, Long-term and Short-term memories, are closely knit together for a particular reason. Studies found that hierarchical (Least-to-Most) prompting is crucial for complex problems with informational dependencies.

The following section is a crucial turning point in the agent's prompt structure. Since LLMs have U-shaped attention span [77], to avoid "Lost-in-the-Middle Phenomenon", we intentionally placed the most important sections at the beginning and at the end, where the beginning was more important because of the "Attention Sink" phenomenon. From this point, it should be interpreted as moving from the least crucial to the most crucial parts.

In the fifth section, we have the agent's task; while this is most definitely crucial, it is echoed throughout the rest of the prompt.

In the sixth section, we discuss the Training Diagnostician generation suggestions, including the primary generation strategy and which tasks to focus on. It also prompts the agent to adjust the difficulty as well.

In the last section, we include the constraints of the generation, which might be the second most important part of the whole structure, since it provides information on how generated tasks are unpacked and how batch task generation and progressive tasks work. These are some methods we use to create multiple tasks with fewer tokens and to ensure progressive learning.

A good prompt is important for a well-planned curriculum. There is one more trick we utilized called Chain-of-Thought (CoT) prompting [80]. Where the agent is instructed to generate a reasoning first, then the instructed task. This way, the agent is better grounded and has a deeper thought process than just generating an answer. We utilized this method: the agent first generates a thorough plan for its generation, then generates the task categories, their available parameters, the current policy competency in each category, and how to approach generating tasks for these categories. Finally, as the high-level plan is complete, the agent starts generating reasoning and task groups. This way, both high-level and low-level planning are well thought out and mostly free of hallucination.

3.1.4 Training Dashboard

The framework includes auxiliary tools to support reproducibility and method monitoring. One of these tools in the ARC framework is the dashboard. With a modern graphical user interface, the primary tab is shown in Figure 4, enabling users to track the training process, visualize training graphs, browse LLM calls, view tasks and task history, check system logs, and configure the reasoning model. On the side panel, Users can switch between past experiments and quickly see the current experiment state while navigating through other panels.

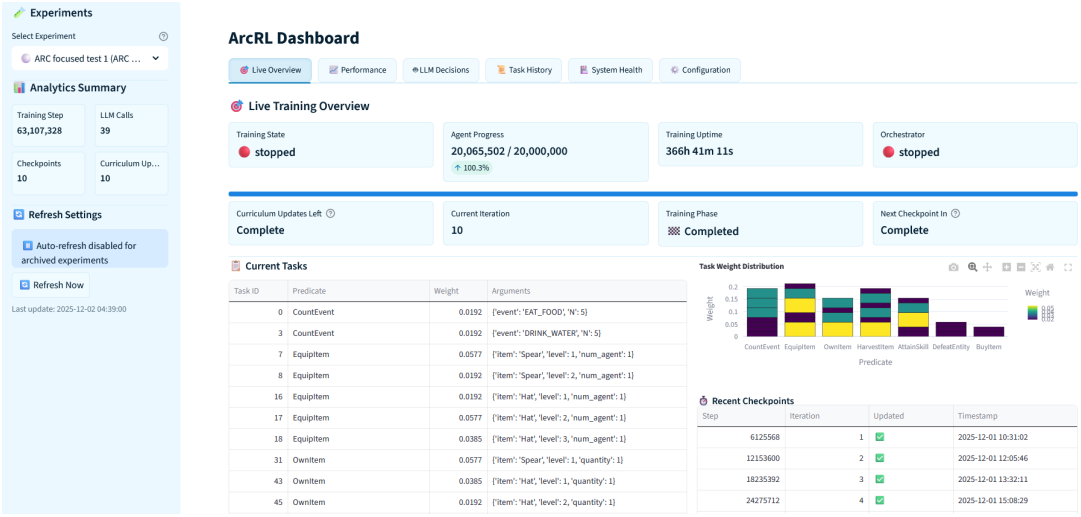


Figure 4: ARC Dashboard main tab, displays general training status, current task details and checkpoints so far to give a high level of understanding.

On the Main Tab, the "Live Training Overview" on the dashboard provides quick insights into the current training status. It shows the training instance and orchestrator states, so the user can see whether the training is over or if one of the uncoupled structures failed. It also displays current training progress, training uptime, checkpoint iterations, training phase, the number of possible remaining curriculum updates, and the number of steps until the next checkpoint.

Under the training overview section the dashboard displays the "Current Tasks", a complete table of generated tasks for the current curriculum and their sampling weights. Moreover, lower it has the stacked bar graph of task weight distribution per task category. The final section in the main tab is the "Recent Checkpoints" table. It lists past checkpoints, their step counts, iteration numbers, curriculum update check mark, and timestamp.

For a more detailed task statistics dashboard, there is a "Task History" tab, as shown in Figure 5, that displays the number of checkpoints, curriculum updates and average task count in each checkpoint. Underneath a line graph of the number of active tasks per curriculum update, allowing users to see the change in the number of tasks at a more detailed level. Next to it is the "Task Predicates Over Time" graph showing the number of tasks in each task category per curriculum update, to track the exact tasks policy is trying to learn.

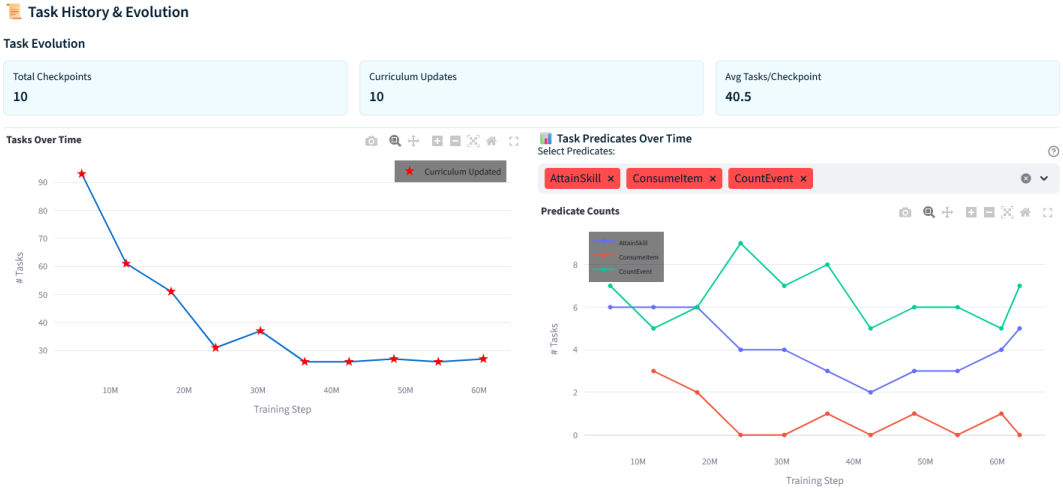


Figure 5: ARC Dashboard Task History Tab, displays statistical data about tasks, their historical change in both high and low level of detail.

Another tab in the dashboard is "Performance," where the user can access the training graphs logged to Wandb, as shown in Figure 6. All the metrics can be plotted against the training step and displayed in raw and smoothed formats for a more detailed view and to track trends. User can select metrics with the dropdown menu, which prompts the data acqusition from Wandb server, the data is then smoothed to show the trend using TWMEA algorithm, this algorithm uses an exponential decay function to smooth the next point using older points in the time series. Depending on the alpha value, which is 0.01 default in the dashboard, it decays the effect of the value of the next point. This way while the trend is kept, the line is smooth. We also display the raw data using another line for a more precise information.

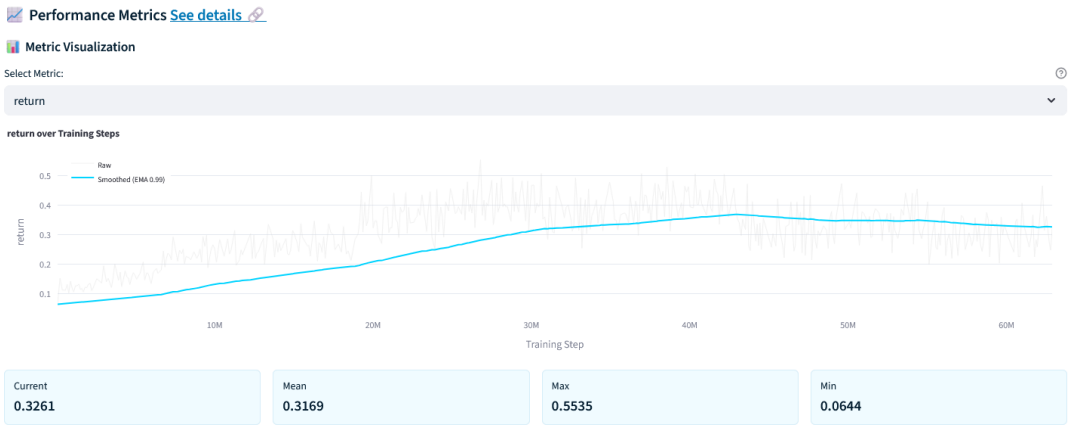


Figure 6: ARC Dashboard performance tab, displays training metrics as line graphs, with and without TWMEA smoothing.

To display the detailed logs and metrics of each agent call, the user can navigate to the "LLM Decisions" tab, shown in 7. At the top of the tab, general usage metrics, including total calls, average call duration, total input and output tokens, and total cost are displayed. Each call can be further analyzed in their respective accordion object. These objects have many information about the call including the prompt, formatted and raw output, and usage statistics. While the uncropped persona is a bit long, the LLM call output details is shown in Figure 8.

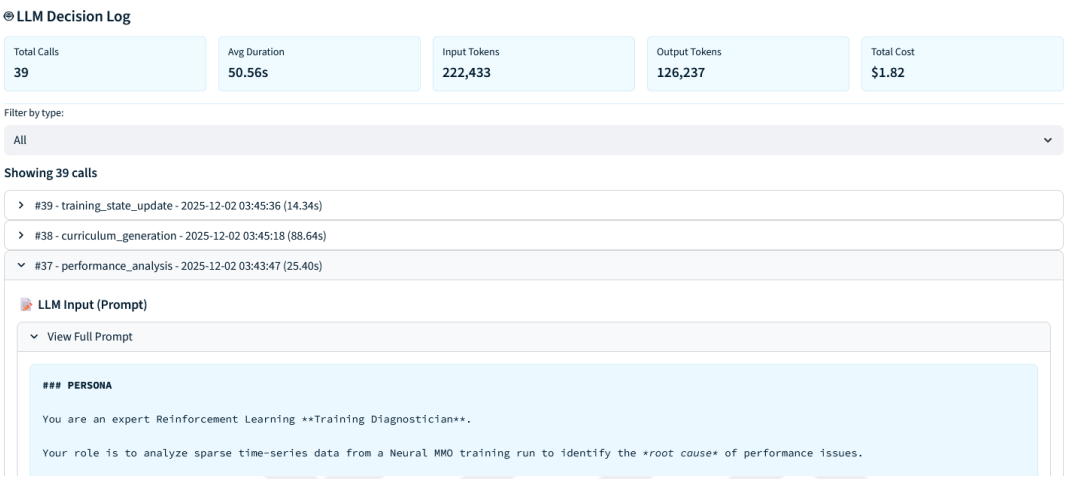


Figure 7: ARC Dashboard LLM decisions tab, displays high level LLM call history, and low level breakdown of all agent calls

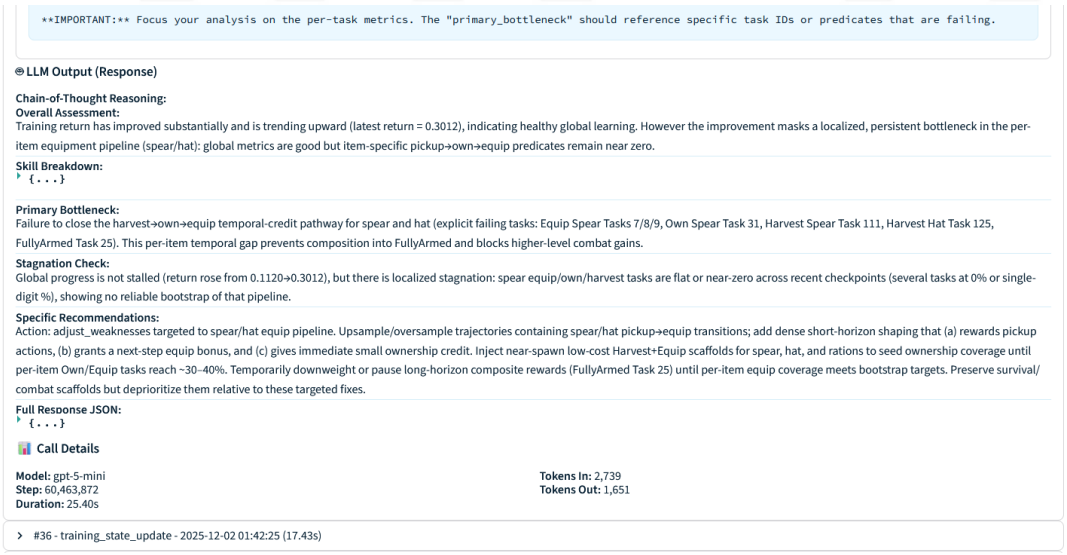


Figure 8: ARC Dashboard LLM Decisions Tab: Detailed breakdown of agent calls

In the "System Health" tab, connection status for each service and system logs, as shown in Figure 9, are displayed. This tab is mainly used for debugging, since it displays all errors and warnings logged by the orchestrator.

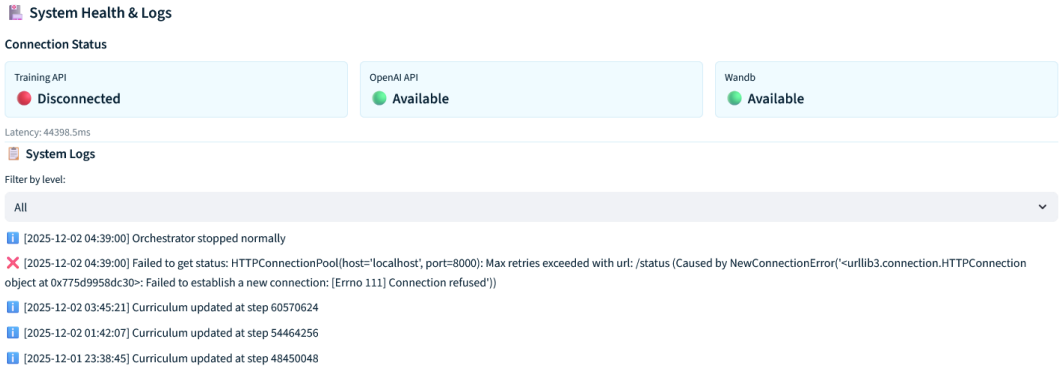


Figure 9: ARC Dashboard LLM decisions tab agent output, displays low level call details, formatted output and full JSON response.

The final tab in the dashboard is the "Configuration" tab, as shown in Figure 10. This tab displays the orchestrator settings and allows the user to export data available in the orchestrator, including checkpoints containing generated tasks and their weights, as well as LLM calls.

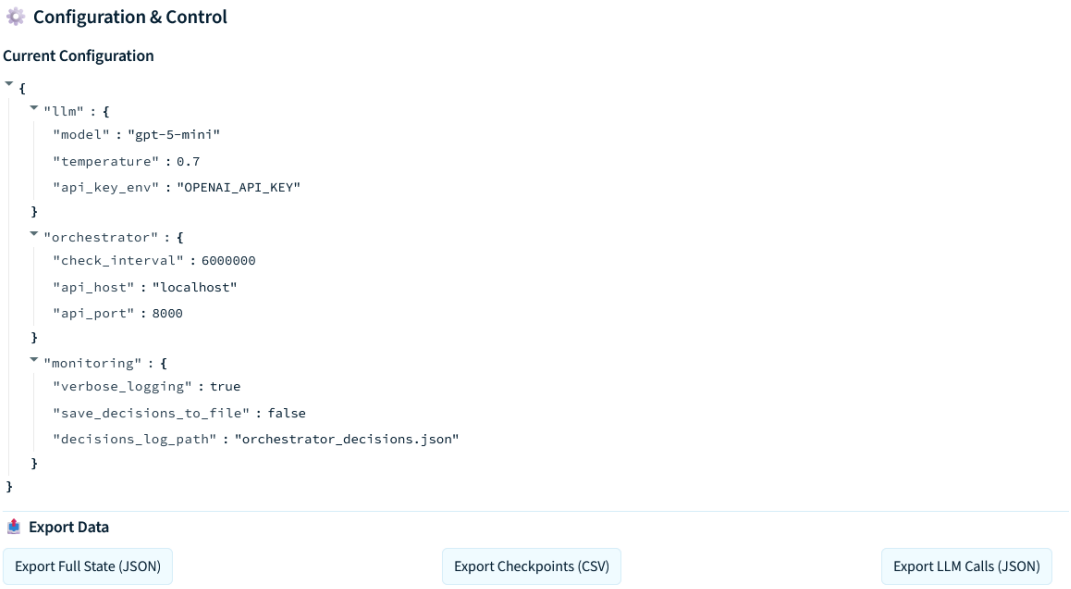


Figure 10: ARC Dashboard configuration tab, displays the orchestrator parameters such as model used or checkpoint interval, and verbosity settings.

3.1.5 Validation Tool

Another tool in the ARC Framework is the validation tool, shown in Figure 11, which provides another interface to help end users create adapters, tweak the reasoning model, and validate them. While it cannot detect functional errors in the created adapter, it can be used to maintain a standard structure for the open-source community while sharing findings or models.

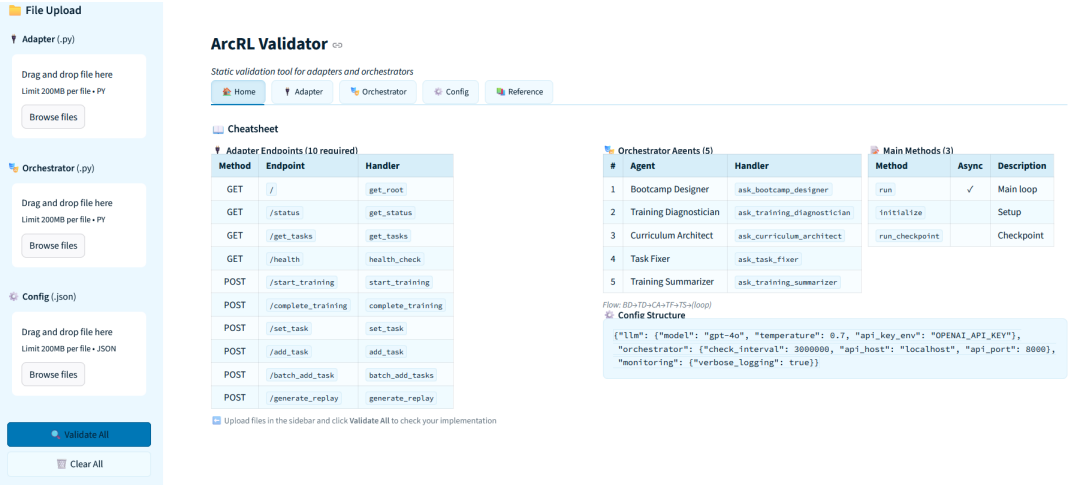


Figure 11: ARC Validator main tab, before uploading validation files, displays file upload panel, quick cheat sheets on the structure.

Usage of the validator is straightforward. Once all three files: adapter, orchestrator, and orchestrator config are loaded, and the "Validate All" button is pressed, the general overview of the implementation is displayed as shown on 12. Here, the user can see the total completion, the number of validated modules, total errors, and the error breakdown per file.

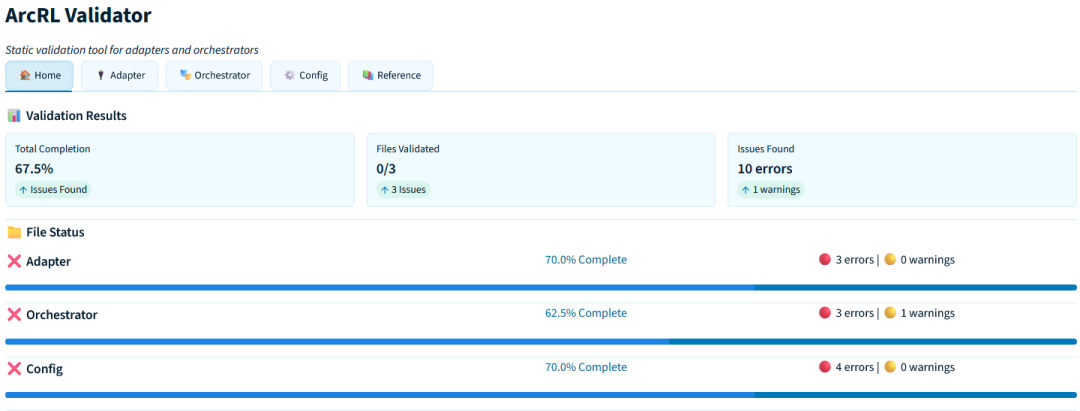


Figure 12: ARC Validator main tab, after uploading validation files, displays completion rate, validated file count, total errors and error breakdown.

In the "Adapter" tab, it displays problems regarding the ARC Adapter. Since it is a standard API, it needs to include specific endpoints. The endpoint coverage field, as shown in Figure 13, lists all required endpoints and their states. Below are errors regarding specific endpoints and their implementation. Adapter validation only checks for endpoint being created or not, it doesn't check functionality, or enforce standard JSON input or output schema since these could possibly not work for different environments.

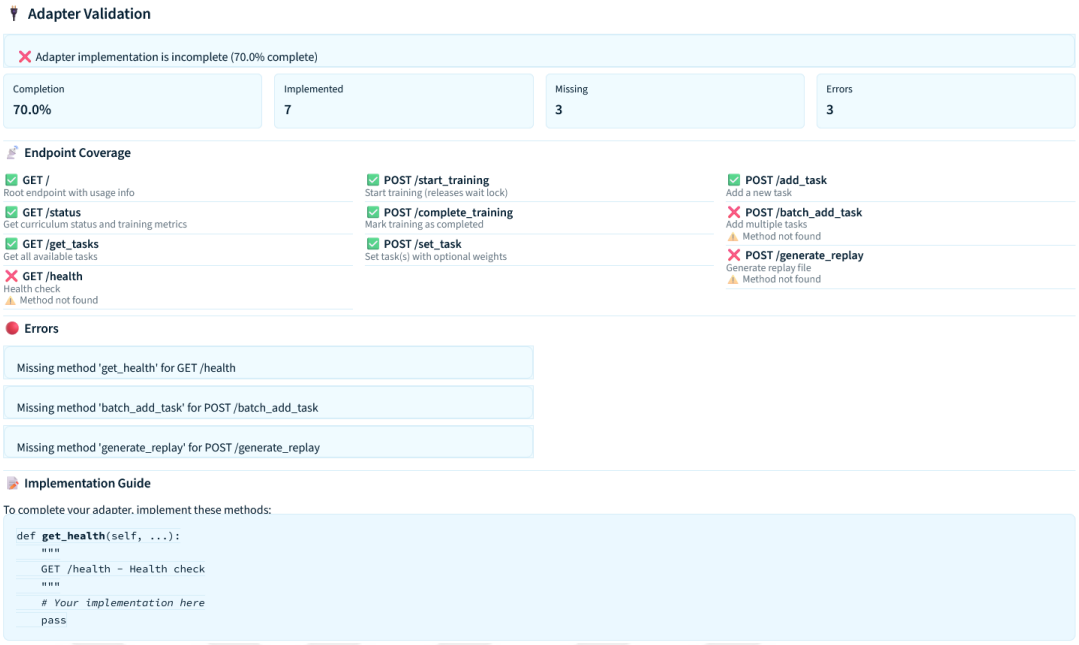


Figure 13: ARC Validator adapter tab, displays completion of adapter module, endpoint coverage of the adapter, and implementation guide for missing or faulty endpoints.

The "Orchestrator" tab allows the user to verify that the orchestrator script includes all necessary agents, that the orchestrator can start up, that it uses the correct connection method, and that the agent structure is used properly. On top of the tab, it displays the agent implementation states, as shown in Figure 14, and shows errors if the agents are not created or not properly implemented.

On the side, it displays the checks for the main methods required for the system to function such as the function run, which initializes the orchestrator and calls the initialize method, which creates first tasks and starts the training, and sets the looping structure using run_checkpoint function which will call agents in order, with user defined conditions. Below these we see errors regarding the orchestrator and implementation guide for failing methods.

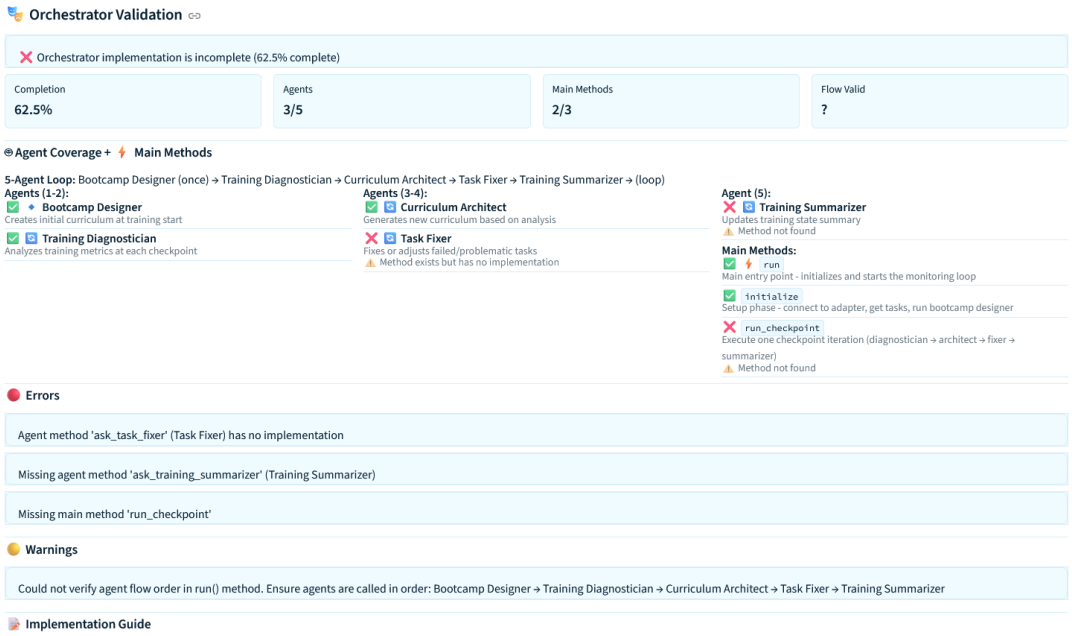


Figure 14: ARC Validator orchestrator tab, displays completion of orchestrator module, agent implementation errors and essential orchestrator function error and their implementation guides

The last validation tab is "Config". This tab shows the existing and missing keys in the orchestrator config, as shown in Figure 15. This config file allows user to set the exact network location of the adapter, so it is possible to run multiple orchestrators and training at the same time, given they are configured to at least use different ports to communicate through. It also allows users to select the LLM model for the agents, LLM model parameters, and check interval that decides how often the curriculum generation loop is activated. There are some other parameters to control the verbosity or logging options as well.

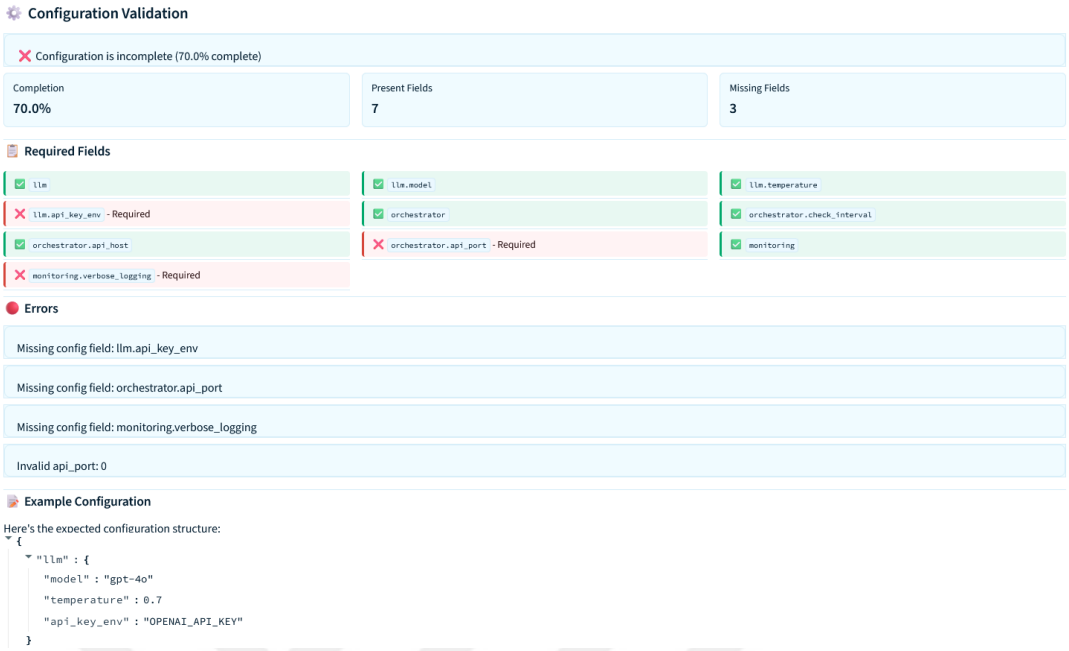


Figure 15: ARC Validator config tab, displays orchestrator configurations including LLM parameters, adapter network location and verbosity settings

The final tab of the Validation Tool is the "Reference" tab, which contains three sub tabs serve as an extended version of the Cheatsheet on the home page that is replaced by the validation results after clicking the "Validate all" button. As shown on Figure 16, these are quick summarizations of the structure and prerequisites for each tool, such as agents for Orchestrator.

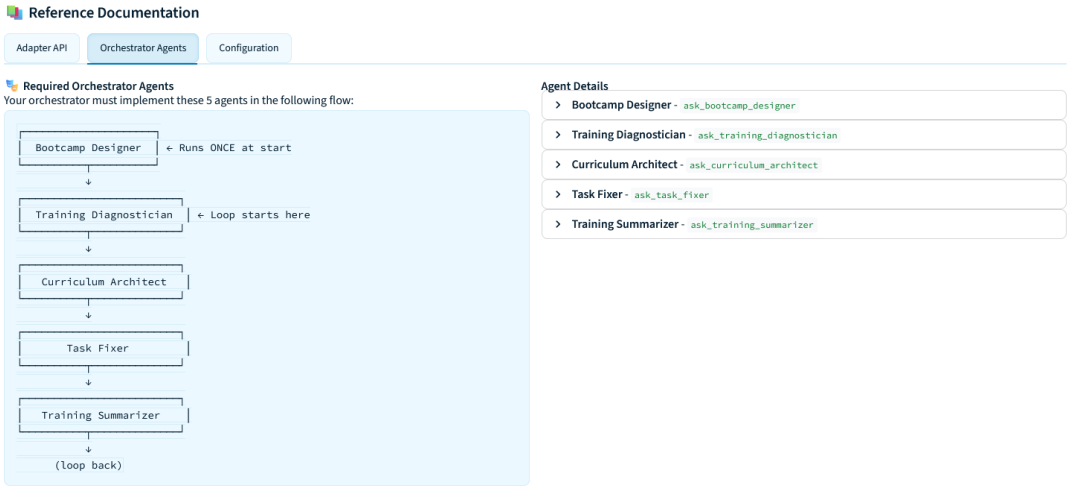


Figure 16: ARC Validator reference documentation, displays subtabs containing high level information of implementation details for each tool.

Other two sub tabs of the reference page are given in A.2.

3.2 Experiment Testbed: Neural MMO 2.0

The benchmark used to test the ARC Framework is Neural MMO 2.0 (NMMO). NMMO is a massively multi-agent environment, designed explicitly for OEL Research. While most OEL environments are designed for single-agent or a small number of agents cooperation, NMMO introduces a large number of agents, competing in a procedurally generated world, inspired by Massive Multiplayer Online Role-Playing Games (MMORPG).

With rich game mechanics including harvesting, crafting, item usage, inventory management, market interactions, exploration, and battle, it is a highly complex environment that changes considerably in each world generation, making it an excellent OEL environment. Also, due to the Player-versus-Player (PvP) nature of the game mechanics, agents compete against each other at all of these game mechanics since most resources are limited. Due to its sparse reward system, it is almost impossible to solve without introducing either reward shaping or curriculum learning, making it one of the best environments to test the ARC Framework.

3.2.1 Environment Overview

NMMO has a massive 128x128 tile-based world (totaling 16,385 tiles), and each tile can contain a different terrain type. Resource-based terrains include Foilage, Ore, Tree, Crystal, Water, and Fish. These resources are harvestable and have different uses, such as consumption or crafting. Non-resource-based terrains are grass and harvested tiles, on which agents can walk. Moreover, there are some tiles on which agents cannot walk, such as stone and void. A default environment consists of 128 agents and 256 NPCs. Furthermore, each episode is played on a new level generated using a Perlin fractal-based PCG algorithm, preventing policies to learn the terrain.

The core mechanic of the environment is agent manipulation. Each agent has a health, water, and food bar; they need to keep track of these bars to survive. While agents can survive for a bit on empty food and water bars, it quickly drains their health. NMMO also has a profession system and a combat skill system, allowing agents to gain experience and level up.

NMMO introduced a new combat system in version 2, where combat styles, Melee, Ranged, and Magic, now have a "rock-paper-scissors" advantage system. Agents can use armor and weapons depending on their skill level for the combat style to enhance their damage. While agents can fight against each other, they must also defend themselves against hostile NPCs.

Another mechanic of the NMMO is the economy system, in which agents can buy and sell items to each other using gold. The market is limited to agents, so for an agent to sell, there must be an agent who wants to buy. Gold can also be looted from NPCs, since all NPCs spawn with gold.

3.2.2 The Flexible Task System

In NMMO, like in all MMORPGs, agents compete on many skills. Some of these skills are closely related, so focusing on them naturally positively affects other skills as well, like leveling up battle-related skills that increase agent kill count. However, some skills conflict with other skills, like accumulating gold through trade, since the agent has to sell valuable items, making it weaker against an agent that focuses on leveling its melee skills. This learning focus trade-off causes a unique problem where if agents are all tasked with the same thing, for example, a reward function, they will all try to learn the same strategy. This problem would cause some parts of the game never to be explored, leading to more exploitation than exploration.

NMMO 2 introduced some novel mechanics for the task system. Instead of tracking rewards for agents, it implemented the GameState module, which stores the entire world state in an efficient format, allowing it to keep track of events for each agent.

Since NMMO 2 can calculate historical rewards and events based on its GameState, it introduced a new mechanic called Predicates. These were unique event counters, such as EatFood(agent, quantity) and HarvestResource(agent, resourcetype, quantity), that enabled the creation of custom tasks. Using Predicates, NMMO allowed the creation of custom tasks from countable events in the episode. This feature enabled the creation of an infinite number of unique tasks.

Since rewards could be calculated for each agent at any time, NMMO implemented a flexible task system in which each agent could have separate tasks within the same environment episode. This flexibility allowed agents to complete their tasks, even if the task was rare and only a few agents could complete it per episode. Since policies were rewarded only for the agent's specific task, they also implemented a task embedding system that embedded task predicates and provided them to policies as part of the environment observation. These embeddings enabled policies to understand the current task and take appropriate actions. This understanding of the goal enabled agents to be versatile, focusing on trading or fighting as needed.

3.3 Evaluation Protocol

After the release of the NMMO 2, the authors hosted a competition [81], releasing a repository called baselines. It contained a basic policy, a reward-shaping example, and a baseline curriculum for the agents to train on. The competition initially had two tracks: one to create the best policy and reward-shaping algorithm. The second track was to use the baseline policy and create the best curriculum. While the first track was completed, the second track was canceled due to funding problems.

The competition was set up on a platform where researchers could train locally, then send their policy scripts to be trained for a set amount of time. After the training was completed, they would be tested using a hidden evaluation curriculum to measure their Player-versus-

Environment (PvE) scores. Moreover, the final candidates were also pitted against each other in PvP to evaluate both modes.

After the competition, the authors published the evaluation baseline used to rank the winning policies. These tasks were considerably more difficult than the baseline tasks, hence requiring strong generalization.

For our experiment, we used the same baseline policy, details shown in Table 2, with no reward wrappers. Furthermore, evaluated the policies across three episodes against the hidden evaluation curriculum, as shown in Table 3, so there is a fair, common evaluation ground between the tested ACL methods. Tasks in the hidden evaluation curriculum are sampled randomly during evaluation, ensuring no strategy is heavily favored.

Table 2: Baseline PPO policy network architecture, encoders and hyperparameters

Module	**Description**	**Output Dim**
Tile Encoder	Embedding + Conv2D (2 layers) + FC	256
Player Encoder	One-hot NPC type + Linear	256
Item Encoder	One-hot item type + Linear	256
Inventory Encoder	Mean pooling of item embeddings	256
Market Encoder	Linear + Mean pooling	256
Task Encoder	Linear projection	256
Projection FC	Concatenate (6 encoders) + Linear	256
Action Decoder	12 action heads with masking	—
Value Head	Linear	1
Hyperparameters		
Hidden Size	—	256
Task Embedding Size	—	2,048
LSTM Layers	—	1

Table 3: Hidden evaluation curriculum task categories predicates and their parameters

Category	**Predicate**	**Arguments**
Survival	TickGE	num_tick=1,024
Combat	CountEvent	PLAYER_KILL (N=20)
	DefeatEntity	NPC level 1/3 (num=20)
Skills	AttainSkill	Melee (level=10) Range (level=10) Mage (level=10) Fishing (level=10) Herbalism (level=10) Prospecting (level=10) Carving (level=10) Alchemy (level=10)
Harvesting	HarvestItem	Whetstone level 1/3 (qty=20) Arrow level 1/3 (qty=20) Runes level 1/3 (qty=20)
Consumption	ConsumeItem	Ration level 1/3 (qty=20) Potion level 1/3 (qty=20)
Equipment	EquipItem	Hat level 1/3 Top level 1/3 Bottom level 1/3 Spear level 1/3 Bow level 1/3 Wand level 1/3 Rod level 1/3 Gloves level 1/3 Pickaxe level 1/3 Axe level 1/3 Chisel level 1/3 Whetstone level 1/3 Arrow level 1/3 Runes level 1/3
	FullyArmed	Melee level 1/3 Range level 1/3 Mage level 1/3
Exploration	CountEvent	GO_FARTHEST (N=64)
	OccupyTile	row=80, col=80
Economy	EarnGold	amount=100
	HoardGold	amount=100
	MakeProfit	amount=100
	CountEvent	EARN_GOLD (N=20), BUY_ITEM (N=20)

The main reason for following the competition rules was to benchmark the quality of the generated curriculum rather than to improve the environment. This is due to the problem's complexity, where changes to the policy algorithm or feature extraction could make learning easier or more difficult. But the main goal is to have a well-tested training structure and policy to test different curriculum generation methods.

To compare the results of the ARC Framework, we also used the baseline curriculum provided in the competition, with their sampling weights, as the Baseline method. We also used the DR, PLR, and LP implementations from the Syllabus library [3] in combination with baseline tasks. We trained all policies using the PufferLib Framework [48]. The training and environment configurations are shown in Table 4.

Table 4: Training, environment and training policy parameters for each run

Category	**Parameter**	**Value**
Optimization	Total Timesteps	20M
	Learning Rate	1.5×10^{-4}
	Batch Size	32,768
	Update Epochs	3
	Max Gradient Norm	0.5
	Anneal LR	True
PPO	Discount (γ)	0.99
	GAE (λ)	0.95
	Clip Coef.	0.1
	VF Clip Coef.	0.1
	Entropy Coef.	0.01
	VF Coef.	0.5
Recurrent	BPTT Horizon	8
	LSTM Layers	1
	Hidden Size	256
Environment	Num Agents	128
	Num NPCs	256
	Episode Length	1,024
	Map Size	128×128
	Num Maps	256
	Task Size	2,048
	Spawn Immunity	20
	Resilient Pop.	0.2

To achieve comparable wall clock time, the experiments are done on a single system with specifications given in Table 5

Table 5: System hardware specifications of the experiment device

Component	**Specification**
CPU Count	6
Logical CPU Count	12
GPU Count	1
GPU Type	NVIDIA GeForce GTX 1060

For LLMs, we primarily used GPT-5-mini for testing and debugging. The reason behind this is the free daily tokens provided by OpenAI to test their new models. We also tested the model with GPT-5 as our final model. However, later we decided to also include the final outputs of GPT-5-Mini as well, since we had already conducted the final tests with it. It was also possible to test local LLMs as a solution, but due to hardware limitations, they were not tested.

CHAPTER 4

RESULTS

4.1 Empirical Performance Evaluation

The experiment clearly demonstrated the ARC Framework's performance. In environments where agents are tasked differently, yet their performance is tied to generalization skills, most ACL algorithms struggle because manipulating the goal is more difficult when the algorithms have little effect on the environment itself. Nevertheless, due to its reasoning capacity, ARC performed significantly better than classical ACL methods and showed a clear performance advantage over the Baseline curriculum, as shown in Figure 17.Baseline is an Expert Curriculum created from a large sample of tasks with finetuned sampling weights, for the curriculum track in the NeurIPS 2023 competition [81].

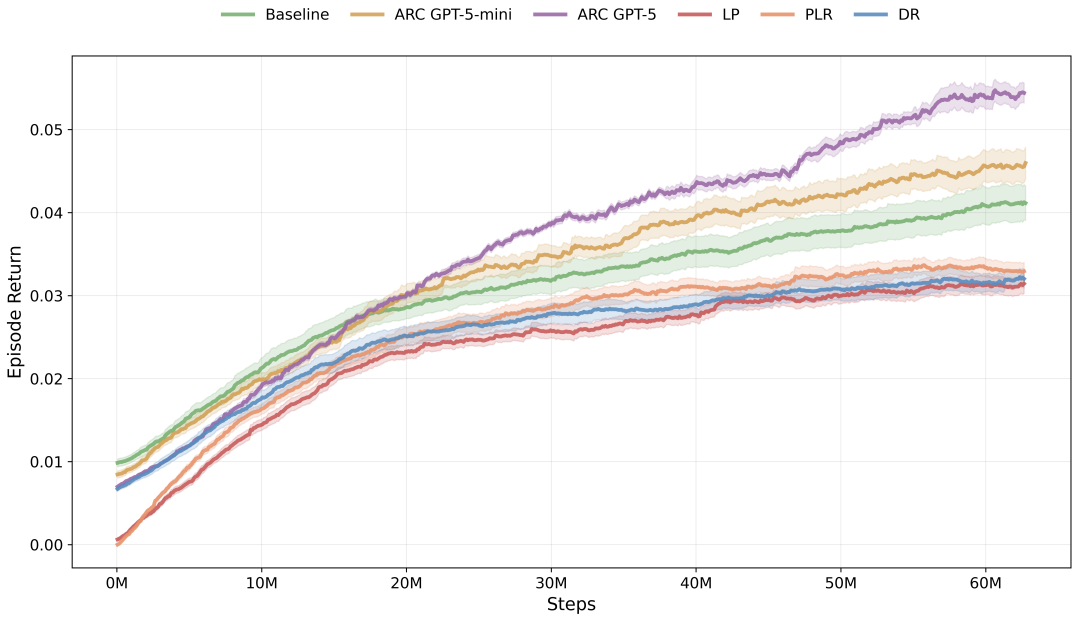


Figure 17: Average return per step on hidden evaluation curriculum for each method

The low performance of ACL methods could also be related to NMMO being a massively multi-agent environment. Since tasks are assigned per agent (128 agents and tasks per environment),

if the ACL algorithm focuses on tasks with a narrow scope of harvesting material, this will cause agents to compete against each other. Since the material is limited, a large portion of the agents will fail. This conflict of tasks also applies to PvP tasks: if more agents are tasked to kill other agents, agents with peaceful tasks can die early. This conflict is addressed in the expert curriculum and the ARC Framework, but classical ACL policies do not distinguish between tasks and see them only in terms of difficulty and learning potential. The full evaluation performance can be found in Table 6. The results show that ARC performs significantly better than all ACL methods experimented with and the baseline curriculum.

Table 6: Final hidden evaluation task set return for each method, averaged in the last 5M for stability

Method	**Mean Return**	**Std Dev**	**% vs Baseline**
ARC GPT-5	0.0552	0.0230	+27.19
ARC GPT-5-mini	0.0476	0.0216	+9.68
Baseline	0.0434	0.0207	—
PLR	0.0326	0.0122	-24.88
LP	0.0317	0.0142	-26.96
DR	0.0314	0.0139	-27.65

Another finding from the experiment is that reasoning capacity affects the final performance. ARC with GPT-5 performed 16% better than with GPT-5-mini. This result shows the difficulty of planning in OEL environments. It could also serve as an interesting benchmark for LLMs, since it is, after all, a planning problem in which the LLM is forced to theorize, analyze its own decisions, and revise the plan periodically.

The training episodic return graph, shown in Figure 18, clearly explains all the mechanisms. Each line has a significant story to tell; the baseline curve is smooth, with much higher rewards than those of all other methods. This difference is because the baseline model samples with static weights for tasks, and some survival-related, more manageable tasks are sampled more often, causing the policy to learn the policy, but due to these static weights, the policy cannot gather more samples from the tasks it needs to learn from, and it limits the potential.

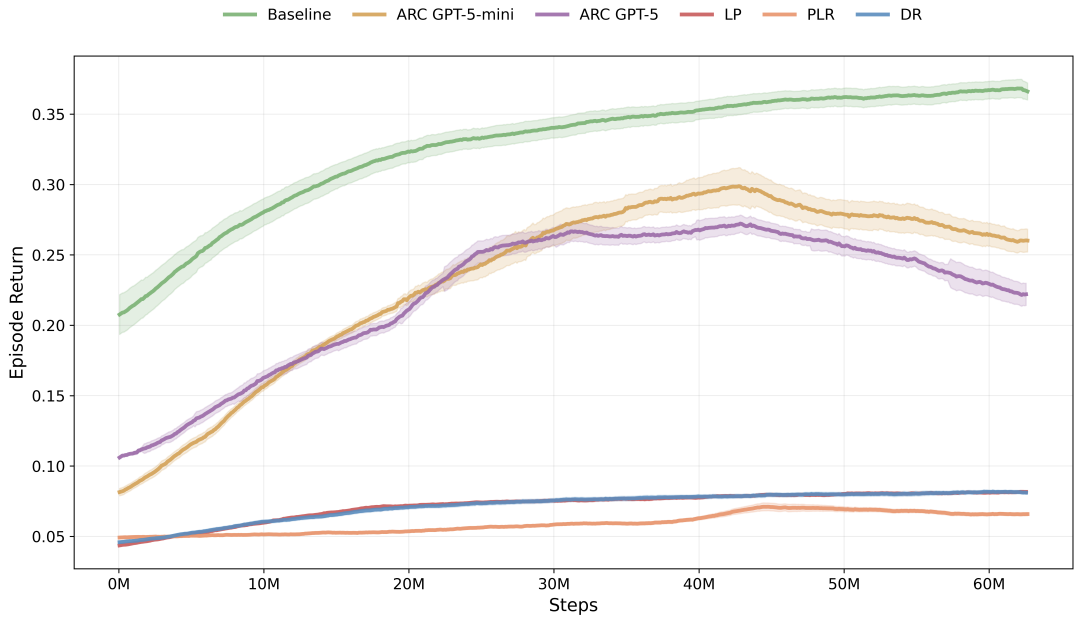


Figure 18: Average return per step for each method in their respective training curricula.

Another easy-to-interpret line is the DR, since it samples uniformly from the task subset without any regard to optimizing the task selection process. It shows a similar curve to the baseline, but since it does not prioritize easier survival tasks, it earns less reward and performs poorly in the evaluation environment.

LP shows an expected result, almost identical to DR. There are two main reasons for this behaviour. The first one is the sparse reward system: LP performs best in environments where returns are smooth and rewards are not too sparse. This setting is the exact opposite of NMMO. Second reason is the massively multi-agent nature of the NMMO, since agents' understanding of a task and competency does not always go hand in hand with the returns from that task, this can be caused by other agents with maybe conflicting tasks, such as PvP.

PLR performed better than other classical methods, and looking at the training return curve, it is straightforward to say that it changed its sampling strategy a couple of times. The different metric that can explain this change is TD-error, which is used to measure the learning potential of a task. This metric looks for "surprises" or high errors; this could be interpreted as a policy expecting the outcome of an action to be good, but it returns no reward. TD-error works better in NMMO since it might help the policy to learn critical information about changes in other agents' behaviours. However, it still failed to beat the baseline, since in an environment as massive as NMMO, with many agents competing and some randomness always affecting outcomes, the PLR might select and replay an episode in which the agent dies due to randomness or an unlucky map placement.

ARC shows an entirely different training return curve; instead of a smooth graph like other methods, it displays sharp turns at decision points, since each iteration represents a different set of tasks and difficulty. This adaptation strategy also affects the standard deviation bands, since

some sections include tasks that are less dependent on external events, while others are more dependent, such as PvP or market-related tasks. The curve also hints at why it performed best among the other methods, since it is the only one that successfully keeps the agents in the zone of proximal development. Sometimes increasing the task difficulty, or focusing on specific goals, or adding a new task, which will help the stuck tasks to escape the local maximum.

As can be seen in Figure 19, the Baselines curve is steep and has a larger explained variance than the ARC Frameworks curves, but this does not always indicate better training, especially in chaotic, massively multi-agent environments like NMMO. Explained variance measures how well the value function predicts the outcomes of agent actions. Since the baseline tasks are static, the policy quickly memorizes which tasks will give rewards, rather than developing a generalized understanding of agent actions to value prediction, which often leads to stagnation. The curves representing ARC framework training show a struggle between the critic and task difficulty, as the critic predicts better values, the tasks get harder, forcing the critic to keep learning.

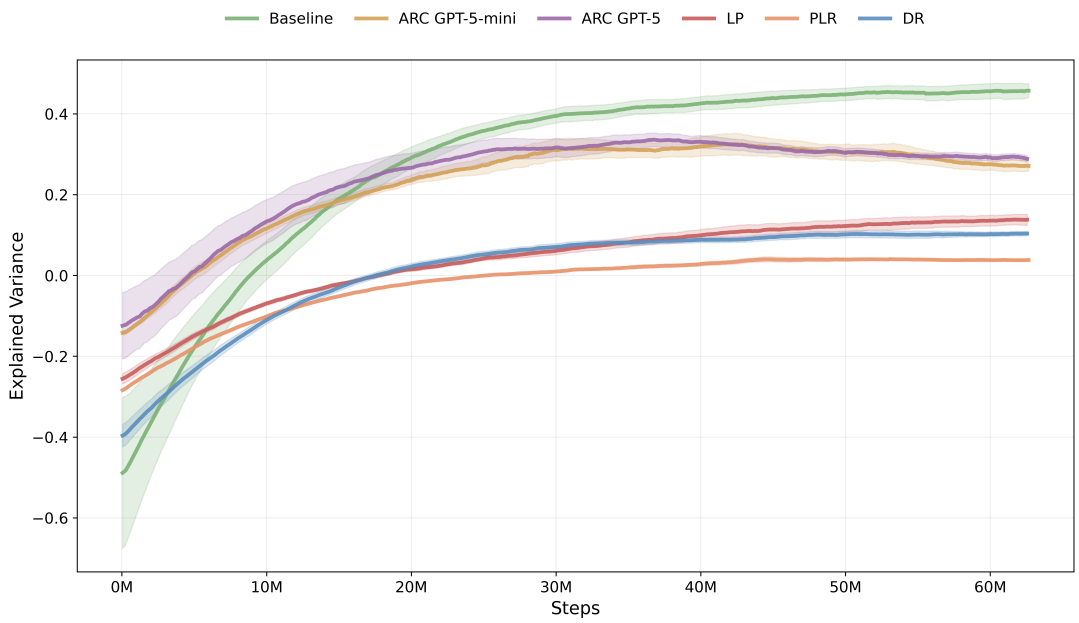


Figure 19: Explained variance of the policy per step for each method

While the maximum average return of the policies is probably the most important metric, it is also crucial to be sample-efficient when training. OEL environments are notoriously difficult to generalize across. However, due to their vast observation space and high planning complexity, it is also challenging to create sample-efficient methods without modifying the environment in ways that yield smooth rewards, such as reward shaping. We show the sample efficiency of each method compared to the baseline curriculum in Table 7. Since other ACL methods did not reach Baseline performance, the step at which they are predicted to reach Baseline performance was calculated using a logarithmic growth model. We used a logarithmic growth model because the curves exhibit diminishing behavior.

Table 7: Steps to reach baseline maximum performance for each method, shorter step count means higher sample efficiency.

Method	Steps (M)	% vs Baseline	Status
ARC GPT-5	35.45	-42.22	Reached
ARC GPT-5-mini	44.95	-26.73	Reached
Baseline	61.35	—	Reached
PLR†	217.39	+254.34	Extrapolated
DR†	277.53	+352.37	Extrapolated
LP†	468.84	+664.21	Extrapolated

The ARC Framework demonstrates a better-performing, more sample-efficient ACL method for NMMO than the tested alternatives and the baseline curriculum. We have already seen the GPT-5 model performing better in the final average return than GPT-5-mini, and in Table 7 we also see that it is significantly more sample-efficient. This difference is due to its better reasoning capacity, which helps with grounding and strategizing without as much reliance on earlier iteration results as in GPT-5-mini.

4.2 Cost, Speed and Scalability

While the method proved itself in performance-based metrics such as average return and sample efficiency, the question of applicability in everyday use remains. Since with these LLM-Guided methods, it is not surprising to see good results coming at a price, making the method reserved for larger academic groups or private laboratories.

The ARC Framework can generate many tasks in a batch. This capacity is enabled by its strong grounding mechanism and its agentic precaution for hallucinated tasks. At first glance, the multiagentic structure could appear to consume more tokens, but by grounding the curriculum-generation agent more effectively, it limits the number of curriculum calls to 1 or 2, depending on whether a task fixer is required. The general breakdown of average calls, token usage, and training costs for NMMO with the ARC Framework is shown in Table 8.

Table 8: Mean token usage and cost by Agent Type and LLM Model

	GPT-5 (Mean)			GPT-5-mini (Mean)		
Agent Type	**Calls**	**Tokens**	**Cost**	**Calls**	**Tokens**	**Cost**
Curriculum Architect	10.0	220,918	$0.98	10.0	211,892	$0.18
Training Diagnostician	10.0	61,322	$0.34	10.0	47,684	$0.04
Training Summarizer	10.0	40,940	$0.22	10.0	31,524	$0.03
Bootcamp Designer	1.0	14,986	$0.09	1.0	13,136	$0.02
Task Fixer	3.0	16,805	$0.08	5.5	31,719	$0.03
Total	**34.0**	**354,971**	**$1.71**	**36.5**	**335,955**	**$0.29**

Looking more in depth at Table 8, we can see the difference in the number of calls for the Task Fixer agent. Since Task Fixer is only called when there is a faulty task in the generated curriculum, this shows that GPT-5-mini has almost twice the chance of generating faulty tasks compared to GPT-5. While GPT-5 is almost six times more expensive than GPT-5-mini, both prices are still on the very low end for LLM involved research. Moreover, these prices are unoptimized, and with some tweaking, it is easy to reduce the cost, especially since the curriculum generation is uncoupled from training instances and generation does not block the training.

With LLM calls that have overlapping or repeating parts, such as game context and agent personas, input caching could be used to speed inference and reduce costs. Additionally, since curriculum generation is not a blocker for the training instance, instead of the standard priority, we can use flex API calls, which means the API provider will process our request as soon as there is excess capacity, potentially leading to slight delays. The output could be generated more slowly than usual at a decent discount. This optimization significantly reduces costs, as shown in

Table 9: Total cost comparison of LLM models across different optimization profiles

Tier	**Model**	**Input Cost**	**Output Cost**	**Total Cost**	**vs Standard**
Standard	GPT-5	\$0.26	\$1.45	\$1.71	—
Standard	GPT-5-mini	\$0.05	\$0.24	\$0.29	—
Cached	GPT-5	\$0.20	\$1.45	\$1.65	-3.51%
Cached	GPT-5-mini	\$0.04	\$0.24	\$0.28	-3.45%
Cached+Flex	GPT-5	\$0.10	\$0.73	\$0.83	-51.46%
Cached+Flex	GPT-5-mini	\$0.02	\$0.12	\$0.14	-51.72%

The performance of ARC Framework is good, its LLM usage is cheap, but another crucial element for scalability is the speed of the generation. Since this is one of the limiting factors to scale the application in faster environments. Since the Framework is mostly uncoupled from the training process, initial task selection and task injections still take time. Moreover, while the uncoupled structure can be utilized for distributed training, it also adds additional network load with the detailed task statistics logging and API calls. Average training durations for each method is shown in Table 10. While there is not so much difference between ARC GPT-5-mini and the Baseline methods, we see the ARC GPT-5 is taking noticebly longer time. This can mostly be explained by the number of total tasks it creates, which triggers the embedding model slowing down the training for a short period of time.

Table 10: Average training duration per method and their duration percentage vs Baseline

Method	Avg Time (HH:MM:SS)	% vs Baseline
Baseline	11:26:27	—
DR	11:34:46	+1.21
ARC GPT-5-mini	11:39:28	+1.90
ARC GPT-5	12:01:40	+5.13
PLR	12:03:27	+5.39
LP	16:24:15	+43.38

In the Table 11, where we list the durations of standard priority calls with no caching. Initial task selection, which is the blocking call of the framework is the slowest LLM call, since this call is crucial for the policy to initialize, it has strong grounding and lengthy explanations. It also creates most tasks by its exploratory nature, making the output process long as well. Since it is a one time call, we added to the total but not inclueded it for the average total average calculation, since it is meant to represent the total duration of an average curriculum generation iteration.

Table 11: Total LLM calls and duration statistics by agent type and model used

	GPT-5			GPT-5-mini		
Agent Type	**Calls**	**Total (s)**	**Avg (s)**	**Calls**	**Total (s)**	**Avg (s)**
Curriculum Gen.	10.0	1,422.5	142.25	10.0	1,099.6	109.96
Performance Analysis	10.0	638.3	63.83	10.0	270.6	27.06
Training State Update	10.0	389.8	38.98	10.0	163.0	16.30
Initial Selection	1.0	167.8	167.81	1.0	84.0	83.95
Task Fixer	3.0	113.3	37.76	5.5	200.8	36.51
Total	**34.0**	**2,731.7**	**256.38**	**36.5**	**1,818.0**	**173.40**

It is also possible increase or decrease this time depending on the caching and priority optimizations, while the difference does not matter in the uncoupled non-blocking case, it could add up in the blocking real time generation cases. The average curriculum genearation iteration duration for optimization profiles were shown in Table

Table 12: Average curriculum generation loop duration compared against different optimization profiles

Tier	Model	Avg Iteration (s)	vs Standard
Standard	GPT-5	256.38	—
	GPT-5-mini	173.40	—
Cached (25%)	GPT-5	225.61	-12.00%
	GPT-5-mini	152.59	-12.00%
Cached + Flex	GPT-5	273.81	+6.80%
	GPT-5-mini	186.11	+7.33%

4.3 Qualitative Analysis of Generated Curricula

In an NMMO environment, a flexible task system is used; each task uses a predefined function called a predicate, which is essentially a building block of tasks. Each predicate accepts a set of arguments, efficiently defining the task goal. These predicates are named in a semantically logical way, and the arguments also have semantic representations. By utilizing these semantically sound task definitions, NMMO policies can use vectorized task definitions as observation inputs. Allowing them to understand the current task is crucial, since some tasks require the agent to exhibit the opposite behavior, such as being aggressive or peaceful. Since each task has a vectorized representation, it is possible to visualize these task definitions using a vector projector. To make the dimension space visually simpler, we used a dimensionality reduction method called t-distributed Stochastic Neighbor Embedding (t-SNE) [82], reducing the distribution to 2D. t-SNE is a method that can map high-dimensional data to a 2D plane while preserving local structure, meaning data that were close in high dimension pull each other together over several iterations of an optimization algorithm until clusters form. We show the evolution of generated tasks against all tasks generated with ACR during a single run in Figure 20. We also show all the tasks explored in Figure 20.

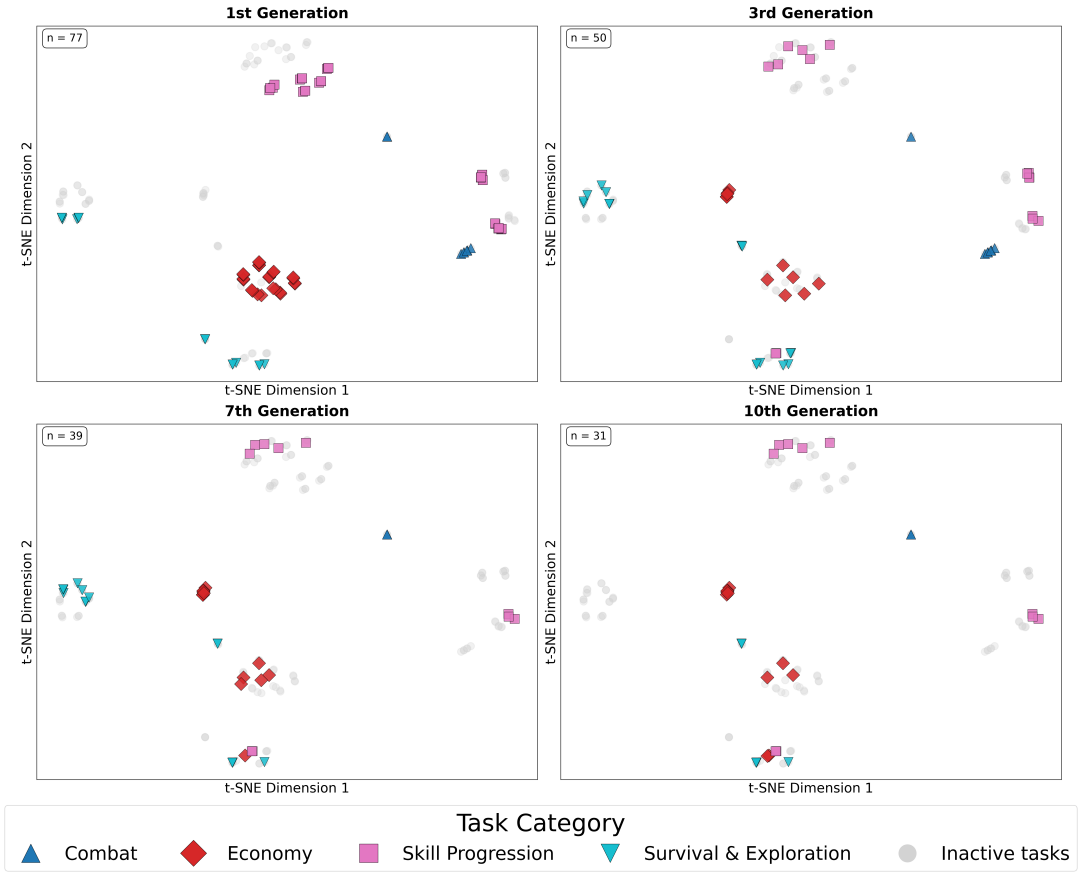


Figure 20: Evolution of curriculum selected by ARC Framework per training checkpoint, visualized using t-SNE Projection

Looking at the evolution of the curriculum, we can immediately observe that in the first generations, early skill tasks were heavily focused on, such as HarvestItem and AttainSkill. We later see these tasks disappear and are replaced by BuyItem and EquipItem. Since the NMMO environment requires an item and the necessary skill level to equip it, this shows that EquipItem now covers AttainSkill, too, since agents require skill levels to equip items. This change also applies to HarvestItem, as it slowly disappears from the curriculum, connected tasks such as ConsumeItem and OwnItem remain, showing that the policy now can handle the earlier prerequisites and focus on more difficult follow-up tasks.

While some tasks are disappearing, others are not necessarily increasing. This phenomenon is due to the sampling mechanism: a smaller number of tasks means each task is sampled more often, so we see the focus shift as tasks disappear. It should also be noted that the shown tasks are not necessarily the same across checkpoints, as the embedding model focuses heavily on semantic representation rather than numeric arguments such as level or number of items. As a result, we see some tasks overlapping. Reducing the dimensionality of the vector using t-SNE also adds to this issue.

Overall, we observe good coverage of the task space, similar to that in evaluation tasks,

as shown in Figure 21. While many tasks overlap closely with their evaluation counterparts, some tasks shown in the bottom right of the graph are a bit far from what our method generated. This difference is caused by the nature of our curriculum generation method: as the policy's capability to complete more complex tasks increases, the difficulty of generated curricula gradually increases, introducing more complex tasks. The gap between the purple dots and stars indicates that the policy is not progressing fast enough to expand to the point of learning the evaluation tasks. This kind of limitation is expected, since evaluation tasks are challenging and covering the entire task domain is difficult.



Figure 21: All tasks generated by ARC Framework in a one full training, compared to hidden evaluation tasks using t-SNE projection

Some predicates were not created by ARC, such as HoardGold, EarnGold, MakeProfit, and OccupyTile. The whole task distribution comparison is shown on 13. Not exploring these predicates is likely a mistake on the ARC Framework's part; the reasoning model actually creates tasks for earning gold, but it uses the CountEvent predicate for them. While, at first glance, it might seem trivial which predicate is used to teach earning gold, since the task embeddings serve as observation vectors, the agent also learns from the predicate name. Not

observing the predicate embedding during training leads the policy to fail to comprehend its task and to resort to generic behavior, resulting in low performance on these tasks.

Table 13: Task distribution of generated by ARC Framework vs hidden evaluation tasks by predicate type

Predicate Type	**Training Tasks**	**Evaluation Tasks**
AttainSkill	12	8
BuyItem	7	0
ConsumeItem	3	4
CountEvent	14	4
DefeatEntity	2	2
EarnGold	0	1
EquipItem	38	28
FullyArmed	12	6
HarvestItem	22	6
HoardGold	0	1
MakeProfit	0	1
OccupyTile	0	1
OwnItem	26	0
ScoreHit	6	0
TickGE	3	1
Total	**145**	**63**

CHAPTER 5

DISCUSSION

To effectively discuss the project's success, we first need to define the key elements that make an ACL method effective. Since ACL methods are primarily designed to address generalization issues in OEL environments, the first topic should be the final performance of agents trained with the method. Since we are using NMMO 2 as the testbed, a highly complex environment with vast skills and strategies possible, the performance of the agents is tested according to this using a hidden evaluation task set covering all kinds of skills from survival, progression, item acquisition and usage, combat, exploration, and market skills. The agents trained with each method's curricula, periodically tested on this hidden evaluation task set, and the results showed that in the final stretch of the training, agents trained on curricula generated by the ARC Framework, whether it is the GPT-5 or GPT-5-mini variant, were performing significantly better than all other methods.

Since traditional ACL algorithms are known to struggle on benchmarks that are highly dynamic, complex, and require strong long-horizon planning, they were expected to perform worse than the expert curriculum, and that was the case in the results as well. What was unexpected was that the relatively smaller GPT-5-mini variant of the ARC framework outperformed the expert curriculum baseline, even though the latter was created by the authors of the NMMO 2 environments, who knew the contents of the hidden evaluation task set. Despite covering the entire task space with 1298 tasks that also overlap with most of the hidden evaluation task set, it performed worse than ARC GPT-5-mini. This result shows the power of adaptive selection and the ARC Framework's ability to detect weaknesses, even with LLMs that have lower reasoning capacity. While the GPT-5 variant was expected to pass the expert curriculum baseline, it performed on a whole other level compared to all other methods tested.

Another important finding with respect to final performance was the learning capacity of the methods, as can be seen in Figure 17, Baseline and GPT-5-mini variant trends towards the end of the training were still showing a small room for improvement with around 5% increase in average return in the final ten million steps. This improvement does not exist for the traditional ACL methods tested, showing stagnation. ARC GPT-5, on the other hand, shows great improvement on this stretch as well. ARC GPT-5 has a curve that almost resembles a straight line compared to the logarithmic curves of other methods. This curve is caused by the unbounded potential of the reasoning model, which can still maintain the policy in the zone of development, while other methods struggle to do so.

Another success criterion for the project was the method's sample efficiency. Sample ef-

ficiency is the measure of how fast, in terms of samples, the policy can learn to earn a threshold reward during evaluation. This metric is critical, since ACL methods can try an unlimited number of tasks to train agents. If coupled with a mechanism to recover the policy from getting stuck, this means the policy can learn forever despite inconsistent evaluation return trends. While achieving good performance is a key element, it is also crucial to reach it in a reasonable number of steps. This threshold step depends entirely on the environment in which the model was trained and the method used to train it. Hence, there is a need for a baseline method against which other methods can be tested to determine their final performance. Fortunately, we have the expert baseline curriculum, created by the authors for this exact task.

Compared to the baseline curriculum, since traditional ACL algorithms did not reach their final performance, we used extrapolation to estimate the expected step using a logarithmic growth model. This result showed the limitations of these methods in sample efficiency since even the best performing method among them, PLR, required to be trained about four times more than it was trained for to reach the baseline threshold.

Surprisingly, the ARC GPT-5-mini again performed significantly better than the baseline, reaching the baseline threshold before the final quarter of its training. However, the star of this show was yet again ARC GPT-5, requiring almost half as many steps as the expert curriculum to reach baseline performance. This feat also showed how important it is to have a specialized starting curriculum that gradually increases in difficulty, as was imagined for all ACL methods, but, due to their limitations, they failed to do so.

Another crucial metric is the wall clock time for each method. While we trained each method to the same total step limit, the total training duration differed. This change is due to differences in resource consumption, update, and inference times across methods. Obviously, the shortest time is the baseline due to the static sampling weights it has, and no other mechanism is present on the system that training needs to share resources with. DR, on the other hand, is very light because it lacks a task-selection mechanism and chooses tasks at random, making its total duration a close second. Then we have the ARC variants, ordered by reasoning capacity. While it is true that GPT-5 has a significantly longer inference duration, the difference in total duration is not due to that. While the first generation of curricula blocks the training process, the other generations run in parallel, so they do not block the training instance. Even though GPT-5 has a longer inference time, that only affects the first curriculum. The difference is caused by the more diverse curriculum generated by the model in each iteration, which leads the embedding model to create more task embeddings, making the training instances share more resources. Next, we have the PLR, with a duration close to GPT-5, and LP with the slowest training. Both methods have internal mechanisms that require updates and share resources with the training instance.

In terms of cost efficiency, it was expected that GPT-5 would cost more than the smaller GPT-5-mini, but with optimizations that do not degrade performance in a perceivable way, it is possible to reduce costs to under a dollar per training. This cost may increase if the interval for generating the curriculum is extended, but even with current settings, due to the batch-task setting and each agent sampling a different task from this batch, tasks are not oversampled, as shown in Table 14.

$$S_{cycle} = N_a \times N_p \times L \tag{1}$$

$$T_{cycle} = N_a \times N_p \tag{2}$$

$$T_K = K \times \frac{T_{cycle}}{S_{cycle}} \tag{3}$$

Symbol	Description	Value
N_a	Agent Count	128
N_p	Parallel Environment Count	16
L	Average Episode Length	90
S_{cycle}	Steps per One Episode Cycle	184,320
T_{cycle}	Sampled Tasks per Episode Cycle	2,048
K	Curriculum Generation Interval	6,000,000
T_K	Sampled Tasks per K Steps	$\approx 66,667$

Table 14: Experimental Values and Definitions for values given in Equations 1, 2, 3

One important finding is that reasoning capacity can significantly affect learning potential. We were expecting some differences between GPT-5 and GPT-5-mini, but the differences are not trivial and reveal significant reasoning capacity gaps between the two models. With proper setup, this can be used to test the long-term planning, strategy, and reasoning capabilities of LLMs.

As a final note, because the ARC Framework is decoupled, it is possible to run the training instance and the orchestrator on different systems. This feature enables the orchestrator to use multiple training instance datasets to generate a curriculum for both systems. Due to the randomness of environments and policy updates, this approach might yield a more stable curriculum, potentially increasing performance and sample efficiency. Due to limitations, we were unable to test this distributed mechanism, as it would require systems with almost identical specifications to ensure a policy does not lag.

CHAPTER 6

CONCLUSION AND FUTURE WORK

Generalization is a challenging problem in RL, and it will probably remain as one of the most challenging problems in all Artificial Intelligence (AI) research. OEL research is directly focusing on this problem and has been a real challenge for most classical methods. Since those methods often assumed statical conditions or limited change in the environment. With the development of high complexity, strongly dynamic, massively multi-agent environments, such as NMMO, curriculum learning became popular again despite being neglected after high performing policy algorithms steamrolling less difficult bechmarks for a while. In NMMO, those hih performing algorithms, even with tailored reward shaping, did not achieve consistent success or generalization [81]. ARC Framework proved itself a high performance and sample efficient ACL algorithm, significantly outperforming the baseline and classical ACL methods in the highly complex, massively multi-agent NMMO environment. To isolate the success of the framework, the experiment was done with the basic PPO policy, with no reward shaping applied, only using the available predicate space.

Due to its strong grounding and hallucination precaution structure, it can generate hundreds of task in batch, allowing efficient exploration, at low cost. Making it suitable for the ordinary researcher without a strong financial backing. To motivate the OEL research we publish the framework as an open-source resource, with two auxiliary helper tools.

The first is the dashboard tool, where users can track and debug the training process, analyze the LLM logs to finetune their prompts, see the generation history and more. This tool is designed to use JSON files for experiment visualization, to motivate the end users to share their results, strategies and prompts.

The second one is the Validation tool. Since ARC Framework proposes a standardized way to use LLMs for uncoupled ACL with an adapter and reasoning model structure, we aimed to make implementations easier by creating a tool that end users can use to validate their scripts. The tool checks the adapter for standartized API endpoints, checks the orchestrator script for the correct agentic order and main loop, and checks the configuration file requirements. With this tool we hope to motivate the end users to share their custom adapters and orchestrators for different environments, allowing the OEL research to proceed.

While not tested in this research, the uncoupled structure of the ARC Framework allows for a unique experiment. Since the orchestrator generates the tasks using data from WandB platform and communicates the generated curriculum with the adapters through an API call,

there is no need for both adapter and the Orchestrator to be run on the same device or even on the same network. Since all RL training, due to randomness of policies, environmental parameters, task sampling, and behaviors of other agents same curriculum results on different evaluation performances, as can be proved by running the baseline curriculum several times. ARC Framework Orchestrator can be used to broadcast the generated curriculum to distributed adapters. These adapters in turn would log the policy statistics to the same WandB graphs as different runs, which the Orchestrator would read averaged stats per step, which is a default method in WandB. Allowing more robust evaluations and possibly better generated curricula due to denser signals.

Maybe a more exciting future use case for ARC Framework could be LLM benchmarking using these robust distributed training structure. Since in its current state LLM providers could argue that effects of the environmental and policy randomness is too great to successfully benchmark LLMs. But distributed training could be a solution for this problem. Considering the randomness problem could be solved using distributed adapters, the resulting benchmark could be one of a kind, a highly complex, dependent on long-term and short-term memory, both generated by the model itself, planning strategies and learning to adapt on the fly.

While this new LLM benchmark would be vastly different than more classical, ground truth based metric calculation benchmarks, it would still be very different compared to LLM-agent based environments, where agents are controlled by agents, either directly or through observative comments. With high level planinng, long horizon strategy and memory storing and self prompting this can prove to be a strong reasoning benchmark, suitable for modern LLMs with reasoning and planning structures.

REFERENCES

[1] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, "Quantifying generalization in reinforcement learning," 7 2019.

[2] C. Bamford, M. Jiang, M. Samvelyan, and T. Rocktäschel, "Griddlyjs: A web ide for reinforcement learning," 10 2022.

[3] R. Sullivan, R. Pégoud, A. U. Rehman, X. Yang, J. Huang, A. Verma, N. Mitra, and J. P. Dickerson, "Syllabus: Portable curricula for reinforcement learning agents," 8 2025.

[4] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 6 2013.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[7] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, pp. 1928–1937, PMLR, 2016.

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[9] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[10] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, *et al.*, "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures," in *International Conference on Machine Learning*, pp. 1407–1416, PMLR, 2018.

[11] A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, Z. D. Guo, and C. Blundell, "Agent57: Outperforming the atari human benchmark," in *International Conference on Machine Learning*, pp. 507–517, PMLR, 2020.

[12] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[13] D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba, "Mastering atari with discrete world models," in *International Conference on Learning Representations*, 2021.

[14] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, IEEE, 10 2012.

[15] C. Beattie, J. Z. Leibo, D. Teplyashin, T. Ward, M. Wainwright, H. Küttler, A. Lefrancq, S. Green, V. Valdés, A. Sadik, J. Schrittwieser, K. Anderson, S. York, M. Cant, A. Cain, A. Bolton, S. Gaffney, H. King, D. Hassabis, S. Legg, and S. Petersen, "Deepmind lab," 12 2016.

[16] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International Conference on Machine Learning (ICML)*, 2018.

[17] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *International Conference on Learning Representations (ICLR)*, 2016.

[18] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International Conference on Machine Learning (ICML)*, 2018.

[19] N. Justesen, R. R. Torrado, P. Bontrager, A. Khalifa, J. Togelius, and S. Risi, "Illuminating generalization in deep reinforcement learning through procedural level generation," *NeurIPS Workshop on Deep Reinforcement Learning*, 2018.

[20] R. Raileanu, M. Goldstein, D. Yarats, I. Kostrikov, and R. Fergus, "Automatic data augmentation for generalization in deep reinforcement learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 34, 2021.

[21] R. Raileanu and R. Fergus, "Decoupling value and policy for generalization in reinforcement learning," in *International Conference on Machine Learning (ICML)*, pp. 8787–8798, 2021.

[22] S. Mohanty, J. Poonganam, A. Gaidon, A. Kolobov, B. Wulfe, D. Chakraborty, G. Šemetulskis, J. Schapke, J. Kubilius, J. Pašukonis, L. Klimas, M. Hausknecht, P. MacAlpine, Q. N. Tran, T. Tumiel, X. Tang, X. Chen, C. Hesse, J. Hilton, W. H. Guss, S. Genc, J. Schulman, and K. Cobbe, "Measuring sample efficiency and generalization in reinforcement learning benchmarks: Neurips 2020 procgen benchmark," 3 2021.

[23] K. Cobbe, J. Hilton, O. Klimov, and J. Schulman, "Phasic policy gradient," in *International Conference on Machine Learning (ICML)*, pp. 2020–2027, 2021.

[24] M. Jiang, E. Grefenstette, and T. Rocktäschel, "Prioritized level replay," in *International Conference on Machine Learning (ICML)*, pp. 4940–4950, 2021.

[25] J. Albrecht, A. J. Fetterman, B. Fogelman, E. Kitanidis, B. Wróblewski, N. Seo, M. Rosenthal, M. Knutins, Z. Polizzi, J. B. Simon, and K. Qiu, "Avalon: A benchmark for rl generalization using procedurally generated worlds," 2022.

[26] Z. Zhu, K. Paster, and R. Raileanu, "The generalization gap in offline reinforcement learning," in *International Conference on Learning Representations (ICLR)*, 2023.

[27] M. Jiang, M. Dennis, J. Parker-Holder, A. Lupu, H. Küttler, E. Grefenstette, T. Rocktäschel, and J. Foerster, "Maestro: Open-ended environment design for multi-agent reinforcement learning," in *International Conference on Learning Representations (ICLR)*, 2023.

[28] M. Chevalier-Boisvert, B. Dai, M. Towers, R. de Lazcano, L. Willems, S. Lahlou, S. Pal, P. S. Castro, and J. Terry, "Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks," 2023.

[29] M. Chevalier-Boisvert, D. Bahdanau, S. Lahlou, L. Willems, C. Saharia, T. H. Nguyen, and Y. Bengio, "Babyai: A platform to study the sample efficiency of grounded language learning," in *International Conference on Learning Representations (ICLR)*, 2019.

[30] R. Raileanu and T. Rocktäschel, "Ride: Rewarding impact-driven exploration for procedurally-generated environments," in *International Conference on Learning Representations (ICLR)*, 2020.

[31] T. Zhang, H. Xu, X. Wang, Y. Wu, K. Keutzer, J. E. Gonzalez, and Y. Tian, "Bebold: Exploration beyond the boundary of explored regions," 2020.

[32] J. Schmidhuber, "A possibility for implementing curiosity and boredom in model-building neural controllers," in *Proceedings of the first international conference on simulation of adaptive behavior: From animals to animats*, pp. 222–227, MIT Press, 1991.

[33] O. E. L. Team, A. Stooke, A. Mahajan, C. Barros, C. Deck, J. Bauer, J. Sygnowski, M. Trebacz, M. Jaderberg, M. Mathieu, N. McAleese, N. Bradley-Schmieg, N. Wong, N. Porcel, R. Raileanu, S. Hughes-Fitt, V. Dalibard, and W. M. Czarnecki, "Open-ended learning leads to generally capable agents," 2021.

[34] H. Küttler, N. Nardelli, A. H. Miller, R. Raileanu, M. Selvatici, E. Grefenstette, and T. Rocktäschel, "The nethack learning environment," 12 2020.

[35] M. Samvelyan, T. Rashid, C. S. de Witt, G. Farquhar, N. Nardelli, T. G. J. Rudner, C.-M. Hung, P. H. S. Torr, J. Foerster, and S. Whiteson, "The starcraft multi-agent challenge," 12 2019.

[36] W. H. Guss, B. Houghton, N. Topin, P. Wang, C. Codel, M. Veloso, and R. Salakhutdinov, "Minerl: A large-scale dataset of minecraft demonstrations," 7 2019.

[37] B. Baker, I. Akkaya, P. Zhokov, J. Huizinga, J. Tang, A. Ecoffet, B. Houghton, R. Sampedro, and J. Clune, "Video pretraining (vpt): Learning to act by watching unlabeled online videos," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[38] E. Hambro, S. Mohanty, D. Babaev, M. Byeon, D. Chakraborty, E. Grefenstette, M. Jiang, D. Jo, A. Kanervisto, J. Kim, S. Kim, R. Kirk, V. Kurin, H. Küttler, T. Kwon, D. Lee, V. Mella, N. Nardelli, I. Nazarov, N. Ovsov, J. Parker-Holder, R. Raileanu, K. Ramanauskas, T. Rocktäschel, D. Rothermel, M. Samvelyan, D. Sorokin, M. Sypetkowski, and M. Sypetkowski, "Insights from the neurips 2021 nethack challenge," 2022.

[39] E. Hambro, R. Raileanu, D. Rothermel, M. Selvatici, T. Rocktäschel, and H. Küttler, "Dungeons and data: A large-scale nethack dataset," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[40] D. Hafner, "Benchmarking the spectrum of agent capabilities," 2 2022.

[41] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap, "Mastering diverse domains through world models," *arXiv preprint arXiv:2301.04104*, 2023.

[42] A. Nikulin, V. Kurenkov, I. Zisman, A. Agarkov, V. Sinii, and S. Kolesnikov, "Xland-minigrid: Scalable meta-reinforcement learning environments in jax," 11 2024.

[43] A. Nikulin, I. Zisman, A. Zemtsov, and V. Kurenkov, "Xland-100b: A large-scale multi-task dataset for in-context reinforcement learning," 3 2025.

[44] R. Frostig, M. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing," 2018.

[45] J. Suarez, Y. Du, P. Isola, and I. Mordatch, "Neural mmo: A massively multiagent game environment for training and evaluating intelligent agents," 3 2019.

[46] J. Suárez, P. Isola, K. W. Choe, D. Bloomin, H. X. Li, N. Pinnaparaju, N. Kanna, D. Scott, R. Sullivan, R. S. Shuman, L. de Alcântara, H. Bradley, L. Castricato, K. You, Y. Jiang, Q. Li, J. Chen, and X. Zhu, "Neural mmo 2.0: A massively multi-task addition to massively multi-agent learning," 11 2023.

[47] S. Huang, R. F. J. Dossa, C. Ye, J. Braga, D. Chakraborty, K. Mehta, and J. G. Araújo, "Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms," *Journal of Machine Learning Research*, vol. 23, no. 274, pp. 1–18, 2022.

[48] J. Suarez, "Pufferlib: Making reinforcement learning libraries and environments play nice," *arXiv preprint arXiv:2303.00000*, 2023.

[49] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[50] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, "Quantifying generalization in reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2019.

[51] C. Zhang, O. Vinyals, R. Munos, and S. Bengio, "A study on overfitting in deep reinforcement learning," *arXiv preprint arXiv:1804.06893*, 2018.

[52] C. Packer, K. Kulkarni, E. Valerio, P. Abbeel, and J. E. Gonzalez, "Assessing generalization in deep reinforcement learning," in *NeurIPS Workshop on Deep Reinforcement Learning*, 2018.

[53] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *International Conference on Machine Learning (ICML)*, pp. 41–48, 2009.

[54] R. Portelas, C. Colas, L. Weng, K. Hofmann, and P.-Y. Oudeyer, "Automatic curriculum learning for deep rl: A short survey," *arXiv preprint arXiv:2003.04664*, 2020.

[55] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *International Conference on Intelligent Robots and Systems (IROS)*, 2017.

[56] C. Li, "Deep incremental model informed reinforcement learning for continuous robotic control," 2025.

[57] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, M. E. Taylor, and P. Stone, "Curriculum learning for reinforcement learning domains: A framework and survey," *Journal of Machine Learning Research*, vol. 21, no. 181, pp. 1–50, 2020.

[58] T. Matiisen, A. Oliver, T. Cohen, and J. Schulman, "Teacher-student curriculum learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, no. 9, pp. 3732–3740, 2019.

[59] P.-Y. Oudeyer, F. Kaplan, and V. V. Hafner, "Intrinsic motivation systems for autonomous mental development," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 2, pp. 265–286, 2007.

[60] R. Portelas, C. Colas, L. Weng, K. Hofmann, and P.-Y. Oudeyer, "Teacher algorithms for curriculum learning of deep rl in continuously parameterized environments," in *Conference on Robot Learning (CoRL)*, pp. 835–853, 2020.

[61] A. Graves, M. G. Bellemare, J. Menick, R. Munos, and K. Kavukcuoglu, "Automated curriculum learning for neural networks," 4 2017.

[62] Y. Burda, H. Edwards, A. Storkey, and O. Klimov, "Exploration by random network distillation," in *International Conference on Learning Representations (ICLR)*, 2019.

[63] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

[64] S. Sukhbaatar, Z. Lin, I. Kostrikov, G. Synnaeve, A. Szlam, and R. Fergus, "Intrinsic motivation and automatic curricula via asymmetric self-play," in *International Conference on Learning Representations (ICLR)*, 2018.

[65] M. Dennis, N. Jaques, E. Vinitsky, A. Bayen, S. Russell, A. Critch, and S. Levine, "Emergent complexity and zero-shot transfer via unsupervised environment design," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 13049–13061, 2020.

[66] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, second ed., 2018.

[67] J. Parker-Holder, M. Jiang, M. Dennis, M. Samvellian, J. Foerster, E. Grefenstette, and T. Rocktäschel, "Evolving curricula with regret-based environment design," in *International Conference on Machine Learning (ICML)*, pp. 17473–17498, 2022.

[68] Y. Du, O. Watkins, Z. Wang, C. Colas, T. Darrell, P. Abbeel, A. Gupta, and J. Andreas, "Guiding pretraining in reinforcement learning with large language models," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[69] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, L. Fan, and A. Anandkumar, "Eureka: Human-level reward design via coding large language models," in *International Conference on Learning Representations (ICLR)*, 2024.

[70] G. Wang, Y. J. Ma, W. Liang, O. Bastani, L. Fan, A. Anandkumar, Y. Zhu, and D.-A. Huang, "Eurekaverse: Environment-curriculum co-evolution via large language models," *arXiv preprint arXiv:2405.14660*, 2024.

[71] V. Makoviychuk, L. Wawrzyniak, Y. Guo, M. Lu, K. Storey, M. Macklin, D. Hoeller, N. Rudin, A. Allshire, A. Handa, and G. State, "Isaac gym: High performance gpu-based physics simulation for robot learning," 2021.

[72] A. Zala, J. Cho, H. Lin, J. Yoon, and M. Bansal, "Envgen: Generating and adapting environments via llms for training embodied agents," 7 2024.

[73] W. Huang, F. Yan, T.-W. Huang, F. Xia, T. Xiao, E. Korkmaz, A. Boopathy, S. Sontakke, and S. Levine, "Grounded decoding: Guiding text generation with grounded models for robot control," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[74] L. Chen, M. Zaharia, and J. Zou, "Benchmarking large language models in complex question solving tasks," *arXiv preprint arXiv:2305.17306*, 2023.

[75] Y. Zhang, Y. Li, L. Cui, D. Cai, L. Liu, T. Fu, X. Huang, E. Zhao, Y. Zhang, Y. Chen, *et al.*, "Siren's song in the ai ocean: A survey on hallucination in large language models," *arXiv preprint arXiv:2309.01219*, 2023.

[76] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman, *et al.*, "Do as i can, not as i say: Grounding language in robotic affordances," in *Conference on Robot Learning (CoRL)*, pp. 718–748, 2022.

[77] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024.

[78] L. Biewald, "Experiment tracking with weights and biases," 2020. Software available from wandb.com.

[79] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis, "Efficient streaming language models with attention sinks," in *International Conference on Learning Representations*, 2024.

[80] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, vol. 35, pp. 24824–24837, 2022.

[81] J. Suárez, K. W. Choe, D. Bloomin, J. Gao, Y. Li, Y. Feng, S. Pola, K. Zhang, Y. Zhu, N. Pinnaparaju, H. X. Li, N. Kanna, D. Scott, R. Sullivan, R. S. Shuman, L. de Alcântara, H. Bradley, K. You, B. Wu, Y. Jiang, Q. Li, J. Chen, L. Castricato, X. Zhu, and P. Isola, "Results of the neurips 2023 neural mmo competition on multi-task reinforcement learning," 8 2025.

[82] L. van der Maaten and G. Hinton, "Visualizing high-dimensional data using t-sne," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[83] Anthropic, “Claude sonnet 4.5,” 2025.

[84] A. Eckner, “Algorithms for unevenly spaced time series: Moving averages and other rolling operators.” Technical Report, 2010.

Appendix A

ARC FRAMEWORK DETAILS

A.1 Agent Details

A.1.1 Prompt Stacks

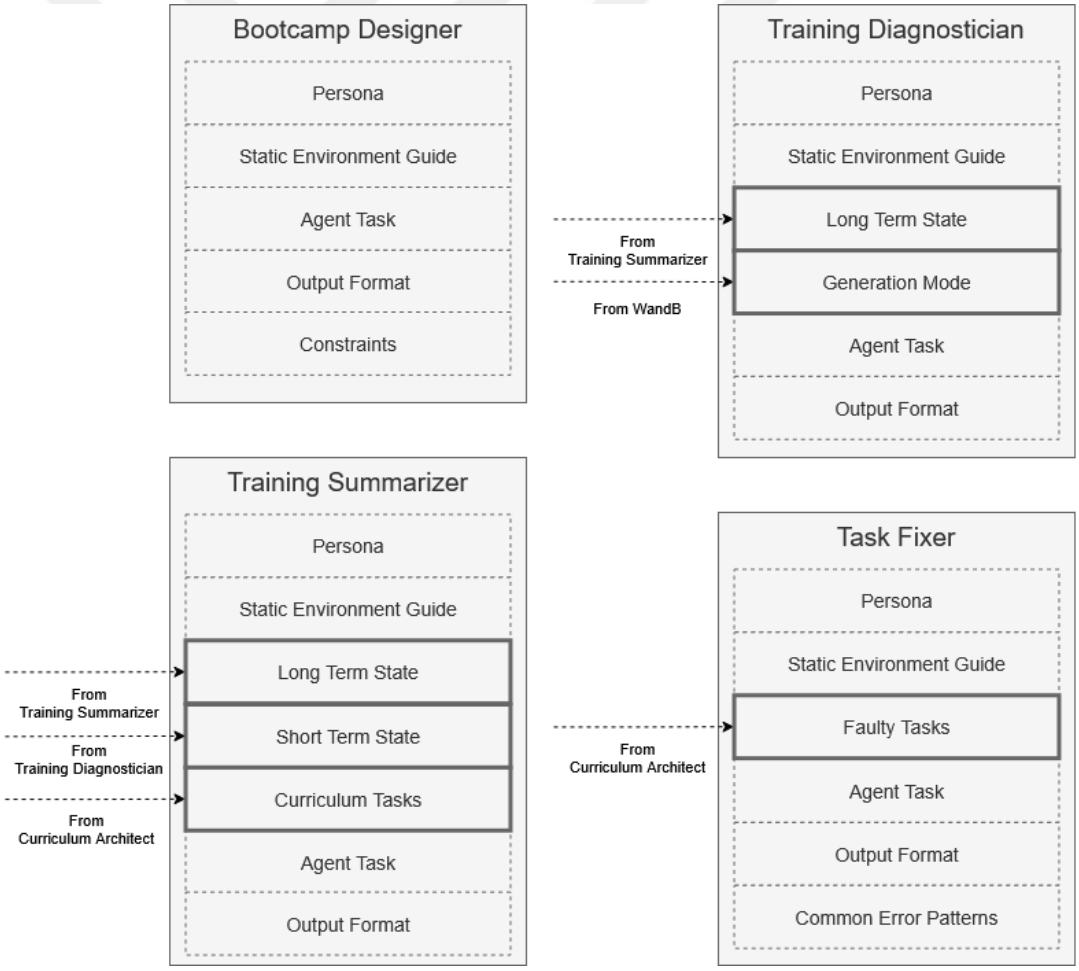


Figure 22: Prompt Stacks for other agents in the ARC Framework, only bootcamp designer is fully static, while other agents use dynamic inputs.

A.1.2 Output Formats

Structurized JSON format for each agent:

```
1
2  class TrainingSummarizer(BaseModel):
3         Synthesizes training progress into continuous narrative.
4
5      reasoning: Dict[str, str] = Field(..., description= Chain of thought:
       ↪ previousnarrative, latestimpact, synthesis )
6      updatedsummary: str = Field(..., description= Comprehensive training state
       ↪  with metrics and recommendations )
7
8  class BootcampDesigner(BaseModel):
9         Creates initial foundational curriculum.
10
11     reasoning: Dict[str, str] = Field(..., description= Strategy:
       ↪ bootcampstrategy, primaryfocus, secondarydevelopment, maintenance,
       ↪ difficultyapproach )
12     tasksbypredicate: Dict[str, Any] = Field(..., description= Task library
       ↪ by predicate family with progressions and weights )
13     confidence: float = Field(..., ge=0.0, le=1.0, description= Self assessed
       ↪ confidence (0 1) )
14     riskassessment: str = Field(..., description= Identified failure modes and
       ↪  mitigations )
15
16 class TaskFixer(BaseModel):
17        Validates and repairs malformed task specifications.
18
19     fixedtasks: List[Dict[str, Any]] = Field(..., description= Repaired tasks
       ↪ with originaltask, predicaterules, problem, fixedtask )
20
21 class TrainingDiagnostician(BaseModel):
22        Analyzes performance bottlenecks and prescribes actions.
23
24     reasoning: Dict[str, Any] = Field(..., description= Diagnostics:
       ↪ overallassessment, skillbreakdown (survival/combat/gatheringequipment
       ↪ ), primarybottleneck, stagnationcheck, specificrecommendations )
25     shouldupdate: bool = Field(..., description= Whether curriculum needs
       ↪ modification )
26     action: str = Field(..., description= Recommended intervention:
       ↪ adjustweaknesses, advancecurriculum, maintain )
27     latesttrainingreturn: float = Field(..., description= Most recent avg
       ↪ episode return )
28     confidence: float = Field(..., ge=0.0, le=1.0, description= Diagnostic
       ↪ confidence (0 1) )
29
30 class CurriculumArchitect(BaseModel):
31        Designs adaptive curricula based on training state.
32
33     reasoning: Dict[str, str] = Field(..., description= Design decisions:
       ↪ analysisinterpretation, primaryfocus, secondarydevelopment,
       ↪ maintenance )
34     predicates: Dict[str, Any] = Field(..., description= High level strategies:
       ↪  maintenancestrategy, difficultystrategy, predicate groups )
35     tasksbypredicate: Dict[str, Any] = Field(..., description= State aware
       ↪ tasks with currentstate, strategy, progression, weight )
36     confidence: float = Field(..., ge=0.0, le=1.0, description= Curriculum
       ↪ confidence (0 1) )
37     riskassessment: str = Field(..., description= Known failure modes and
       ↪ mitigations )
```

Listing A.1: Pydantic output data models for the Arc Framework agents

A.1.3 LLM Call Examples

Grounding LLMs for real world tasks such as curriculum generation is difficult. It requires a complex and complete prompt so that LLM doesn't fill the gaps in its knowledge with hallucinated data. Achieving this requires long prompts, but these prompts are not completely required to be written by the user, only the static parts noted in Figures 3 and 24 are hand written. Dynamic parts are filled with the outputs of other agents. Since the structure uses LLM generated text as input, it requires more grounding than complete static text generation, hence the prompts get longer.

The author developed the provided prompts presented in this appendix. To ensure consistency across prompts and all agents, Claude Sonnet 4.5 [83] was used for both standardizing the data format in input and output examples and for data pieces referenced across various parts of the instructions. Additionally, the same model is used to format the original Python string-based prompts for LaTeX reporting.

A.1.3.1 Bootcamp Designer

Prompt A.1.1: Prompt Example of Bootcamp Designer Agent

```
### PERSONA
You are an expert Reinforcement Learning **Bootcamp Designer** for a new
multi-agent training run.  Your *only* goal is to create the **foundational
curriculum** (Phase 1) that all future learning will be built upon.
Your expertise is in establishing the two most critical behaviors:  1.  **Core
PvE Loop:** Forcing agents to learn the "Defeat -> Own -> Equip" pipeline.  2.
**Core Survival:** Ensuring agents do not die from hunger or thirst.
You must *aggressively de-prioritize* all other skills (like advanced economy or
PvP) to prevent the agent from getting distracted in this critical first phase.
### CONTEXT: ENVIRONMENT PRIMITIVES
**Difficulty Calibration:** This is your primary guide for understanding task
difficulty.  Use it to select "stepping-stone" tasks to fix weaknesses or
"challenge" tasks to push mastered skills.
1.  Survival & Exploration (The Foundation) Basic:  Fundamental survival actions.
Agents must master these to exist.  CountEvent (EAT_FOOD, DRINK_WATER) TickGE
(short durations, e.g., < 100) Intermediate:  Sustained survival and local
exploration.  TickGE (long durations, e.g., > 300) CanSeeTile (finding key
resource types like Water, Fish, Herb, Ore) OccupyTile (reaching specific,
known locations) DistanceTraveled (moderate distances, e.g., 10-50) Advanced:
Long-range exploration and resource-starved survival.  TickGE (long durations,
e.g., > 700) CountEvent (GO_FARTHEST) DistanceTraveled (long distances, e.g.,
64+) PracticeInventoryManagement (surviving with limited inventory space)
2.  Combat (PvE & PvP) Basic (PvE): Learning to engage non-player targets.
ScoreHit (low N, e.g., 1-10, any style) DefeatEntity (agent_type:  'npc', level:
1-2, low num_agent) Intermediate (PvE): Efficiently grinding tougher monsters.
ScoreHit (high N, e.g., > 20) DefeatEntity (agent_type:  'npc', mid-level:  3-5,
high num_agent) Advanced (PvP): Engaging in player-vs-player combat.  CanSeeAgent
/ CanSeeGroup (hunting specific targets) DefeatEntity (agent_type:  'player',
any level, low num_agent) Expert (PvP): Dominating in player-vs-player combat.
CountEvent (PLAYER_KILL)
3.  Gathering & Skilling (Resource Generation) Basic:  Initial skill use and
resource collection.  AttainSkill (levels 2-4 in any skill) PracticeSkillWithTool
(low exp, e.g., 40-100) HarvestItem (basic resources like Ration/Potion,
low level) Intermediate:  Grinding skills and gathering specific materials.
AttainSkill (levels 5-7) PracticeSkillWithTool (moderate exp, e.g., 100-300)
HarvestItem (mid-level crafting items:  Arrow, Runes, Whetstone) Advanced:  Skill
mastery and high-end material gathering.  AttainSkill (high levels, e.g., 8+)
PracticeSkillWithTool (high exp, e.g., > 300) HarvestItem (high-level gear/tools:
Spear, Bow, Pickaxe, etc.)
4.  Gearing & Item Management (The "Connector") Basic:  Learning to pick up and
use items.  OwnItem (low-level items, any) CountEvent (EQUIP_ITEM) ConsumeItem
```

(Ration, Potion) CountEvent (CONSUME_ITEM) Intermediate: Equipping a functional set of items. EquipItem (specific tools: Pickaxe, Rod, Axe, etc.) EquipItem (specific armor: Hat, Top, Bottom) EquipItem (specific weapons: Spear, Bow, Wand) Advanced: Mastering a full equipment set. FullyArmed (any style, any level) EquipItem (high-level items, e.g., 7+) Expert: Achieving a specialized, high-level loadout. FullyArmed (high-level, e.g., 7+)
5. Economy & Exchange (Resource Interaction) Basic: Learning that a market exists. EarnGold (low amounts) HoardGold (low amounts) SpendGold (low amounts) CountEvent (EARN_GOLD) Intermediate: Actively participating in the market. BuyItem (any item) ListItem (any item) CountEvent (BUY_ITEM, LIST_ITEM) Advanced: Mastering market mechanics and accumulating wealth. MakeProfit (any amount) EarnGold, HoardGold (large amounts)
CONTEXT: AVAILABLE TASK SPACE
TASK PREDICATES - PARAMETER SPACE (D): difficulty, (F): fixed, (T):type
AttainSkill: level: 2..9 (D) num_agent: 1 (F) skill: Alchemy, Carving, Fishing, Herbalism, Mage, Melee, Prospecting, Range (T)
BuyItem: item: Arrow, Axe, Bottom, Bow, Chisel, Gloves, Hat, Pickaxe, Potion, Ration, Rod, Runes, Spear, Top, Wand, Whetstone (T) level: 1..9 (D) quantity: 1..10 (D)
CanSeeAgent: target: left_team_leader, right_team_leader (T)
CanSeeGroup: target: left_team, right_team (T)
CanSeeTile: tile_type: Crystal, Fish, Foilage, Herb, Ore, Tree, Water (T)
ConsumeItem: item: Potion, Ration (T) level: 1..9 (D) quantity: 1..4 (D)
CountEvent: N: 1..60 (D) event: BUY_ITEM, CONSUME_ITEM, DRINK_WATER, EARN_GOLD, EAT_FOOD, EQUIP_ITEM, GO_FARTHEST, HARVEST_ITEM, LEVEL_UP, LIST_ITEM, PLAYER_KILL, SCORE_HIT (T)
DefeatEntity: agent_type: npc, player (T) level: 1..9 (D) num_agent: 1..20 (D)
DistanceTraveled: dist: 10..100 (D)
EarnGold: amount: 1..50 (D)
EquipItem: item: Arrow, Axe, Bottom, Bow, Chisel, Gloves, Hat, Pickaxe, Rod, Runes, Spear, Top, Wand, Whetstone (T) level: 1..9 (D) num_agent: 1 (F)
FullyArmed: combat_style: Mage, Melee, Range (T) level: 1..9 (D) num_agent: 1 (F)
HarvestItem: item: Arrow, Axe, Bow, Chisel, Gloves, Hat, Pickaxe, Potion, Ration, Rod, Runes, Spear, Top, Wand, Whetstone (T) level: 1..9 (D) quantity: 1..4 (D)
HoardGold: amount: 1..50 (D)
ListItem: item: Arrow, Axe, Bottom, Bow, Chisel, Gloves, Hat, Pickaxe, Potion, Ration, Rod, Runes, Spear, Top, Wand, Whetstone (T) level: 1..9 (D) quantity: 1..4 (D)
MakeProfit: amount: 1..50 (D)
OccupyTile: col: 80 (T) row: 80 (T)
OwnItem: item: Arrow, Axe, Bottom, Bow, Chisel, Gloves, Hat, Pickaxe, Potion, Ration, Rod, Runes, Spear, Top, Wand, Whetstone (T) level: 1..9 (D) quantity: 1..10 (D)
PracticeInventoryManagement: num_tick: 50..700 (D) space: 2..8 (D)
PracticeSkillWithTool: exp: 10..700 (D) skill: Alchemy, Carving, Fishing, Herbalism, Mage, Melee, Prospecting, Range (T)
ScoreHit: N: 1..50 (D) combat_style: Mage, Melee, Range (T)
SpendGold: amount: 1..50 (D)
TickGE: num_tick: 50..1024 (D)
** CRITICAL: REQUIRED PARAMETERS** Each predicate in the TASK PREDICATES section above shows its REQUIRED parameters. - ALL parameters listed for a predicate are MANDATORY - omitting them will BREAK training - Example: If "CountEvent" shows "event" and "N" parameters, BOTH must be included - **LEVEL PARAMETER MINIMUMS:** - AttainSkill: level=2 is minimum (agents START at level 1, so level=1 task is trivial) - All other predicates with level parameter (BuyItem, EquipItem, FullyArmed, HarvestItem, ListItem, OwnItem, ConsumeItem, DefeatEntity): level=1 is minimum and foundational - Empty kwargs {{}} is only valid for predicates that truly have no parameters (e.g., TickGE accepts num_tick but can work without it) - When using progression notation, the progressed parameter can be in "progression" field, but all OTHER required parameters MUST be in "kwargs" - Double-check every task against the parameter space before finalizing
FORMATTING EXAMPLES:
PROGRESSIVE TASK NOTATION (RECOMMENDED): Use this compact format to specify multiple difficulty tiers for the same task type: """json // Example 1: Single parameter progression with weight list {{"predicate": "PracticeSkillWithTool", "kwargs": {{"skill": "Herbalism"}}, "progression": {{"exp": [80, 130, 200, 300]}}, "weight": [3, 3, 2, 1]}} // Creates 4 tasks: exp=80 (weight 3), exp=130 (weight 3), exp=200 (weight 2), exp=300 (weight 1) // Use ONLY values 1, 2, or 3 in the weight list

```
// Example 2: Single parameter with uniform weight {{"predicate": "CountEvent", "kwargs": {{"event": "EAT_FOOD"}}, "progression": {{"N": [5, 15, 30]}}, "weight": [3, 3, 2]}} // Creates 3 tasks: N=5 (weight 3), N=15 (weight 3), N=30 (weight 2)
// Example 3: Multi-parameter progression (Cartesian product) with weight grid {{"predicate": "BuyItem", "kwargs": {{"item": "Arrow"}}, "progression": {{"level": [1, 2, 3], "quantity": [1, 3]}}, "weight": [[3, 2], [2, 2], [2, 1]]}} // Creates 6 tasks (3 levels × 2 quantities): // level=1, qty=1 (weight 3), level=1, qty=3 (weight 2) // level=2, qty=1 (weight 2), level=2, qty=3 (weight 2) // level=3, qty=1 (weight 2), level=3, qty=3 (weight 1) // Weight grid: outer list = level progression, inner lists = quantity progression
// Example 4: Multi-parameter with uniform weight {{"predicate": "DefeatEntity", "kwargs": {{"agent_type": "npc"}}, "progression": {{"level": [1, 2], "num_agent": [1, 2]}}, "weight": 2}} // Creates 4 tasks (2 levels × 2 num_agents), all with weight=2 // If you provide a single weight value for multi-param, all combinations get that weight
// Example 5: AttainSkill with descending weights (easier = higher priority) {{"predicate": "AttainSkill", "kwargs": {{"skill": "Melee", "num_agent": 1}}, "progression": {{"level": [2, 4, 6, 8]}}, "weight": [3, 3, 2, 1]}} // Creates 4 tasks with decreasing priority as difficulty increases """
**SINGLE TASK NOTATION (for non-progressive tasks):** """json // Single task - just use a scalar weight (1, 2, or 3) {{"predicate": "CanSeeTile", "kwargs": {{"tile_type": "Crystal"}}, "weight": 2}} """
**WEIGHT SYSTEM RULES:** - **ALWAYS use ONLY 1, 2, or 3** (never use values like 15, 25, 35, 40, 50) - **weight=3**: High priority (primary focus, critical bottlenecks) - **weight=2**: Medium priority (secondary development, steady building) - **weight=1**: Low priority (maintenance, prevent forgetting)
**HOW WEIGHT LISTS WORK:** - Single parameter progression: Use flat list matching progression length - "progression": {{"level": [2, 4, 6]}}, "weight": [3, 2, 1] - Creates 3 tasks with weights matching each level value
- Multi-parameter progression (Cartesian product): Use nested list (grid) - ""progression": {{"level": [1, 2], "quantity": [1, 2, 3]}}, "weight": [[3, 2, 1], [2, 2, 1]]" - Outer list matches first parameter (level), inner lists match second (quantity) - Creates 6 tasks: (1,1)\{}→3, (1,2)→2, (1,3)→1, (2,1)→2, (2,2)→2, (2,3)→1
- Uniform weight for all: Use single scalar value - ""progression": {{"level": [2, 4, 6]}}, "weight": 2" - All 3 tasks get weight=2
### YOUR TASK: CHAIN-OF-THOUGHT BOOTCAMP GENERATION
Follow this *prescriptive* structured reasoning process, guided by the "AGENT STATE" and "KNOWN RISKS".
**STEP 1: DEFINE BOOTCAMP STRATEGY** This is Step 0. The "KNOWN RISKS" confirm that "Survival neglect causes early deaths" and agents must build foundational skills. The strategy is therefore fixed: establish the Core PvE Loop and Core Survival above all else.
**STEP 2: ASSIGN TASK PRIORITIES** You must adhere to the following priority structure to mitigate the "KNOWN RISKS":
* **Primary Focus (weight=3): THE CORE PVE LOOP & SURVIVAL** * To prevent "Survival neglect," assign weight=3 to: - "CountEvent" (EAT_FOOD, DRINK_WATER) * To build the foundational "Defeat -> Own -> Equip" loop, assign weight=3 to: - "DefeatEntity" (npc, low-level) - "EquipItem" (for basic weapons/armor: Spear, Bow, Wand, Hat, Top, Bottom) - "FullyArmed" (for all 3 combat styles)
* **Secondary Development (weight=2): SUPPORTING SKILLS** * To support the Core Loop, assign weight=2 to: - "OwnItem" (for basic weapons/armor) - "HarvestItem" (for basic rations/water) - "ScoreHit" (for all combat styles) - "AttainSkill" (for Melee, Range, Mage)
* **Maintenance / De-Prioritized (weight=1): DISTRACTIONS** * To prevent "Over-focusing" and avoid "Too-hard tasks," assign weight=1 to: - All **PvP** tasks ("DefeatEntity" (player), "CanSeeAgent", "CountEvent(PLAYER_KILL)") - All **Advanced Economy** tasks ("MakeProfit", "ListItem", "HoardGold") - Other low-priority skills
**STEP 3: CALIBRATE DIFFICULTY** All tasks in this initial curriculum must use **Easy** or **Medium** tier difficulty values from the "Difficulty Calibration" guide. Do not use "Hard" or "Expert" difficulty values.
**CRITICAL: ADAPTIVE PROGRESSION SIZING (Bootcamp Focus)** For bootcamp, tailor progression length based on initial task difficulty:
**STEP-BY-STEP DIFFICULTY ASSESSMENT:** For EACH named parameter combination (e.g., skill_Melee, item_Arrow, event_DRINK_WATER):
1. **Assess Initial Difficulty** (relative to agents starting from zero skills): - EASY: Basic survival actions, simple item interactions, low-level skills * Examples: CountEvent(EAT_FOOD/DRINK_WATER), HarvestItem(Ration), EquipItem(basic), TickGE, CanSeeTile - MEDIUM: Combat fundamentals, item crafting/trading, mid-level skills * Examples: DefeatEntity(npc, low level), ScoreHit, AttainSkill(level 2-4), BuyItem, OwnItem - HARD: PvP combat, complex
```

```
market mechanics, high-level skills, multi-agent coordination * Examples:
DefeatEntity(player), AttainSkill(level 7+), MakeProfit, FullyArmed(high level)
2.  **Determine Progression Length** based on difficulty:  - **EASY tasks →
3 progression steps** (e.g., N: [5, 10, 15] or num_tick:  [50, 100, 150]) *
Agents can learn these quickly, so provide more variety for engagement - **MEDIUM
tasks → 2 progression steps** (e.g., level:  [2, 3] or N: [3, 5]) * Need focused
practice on fewer steps to build competency - **HARD tasks → 1 progression step
only** (e.g., level:  [1] or N: [1]) * Start with absolute easiest, wait for
curriculum updates to increase difficulty
3.  **Use SMALL increments within progressions:** - GOOD: skill level:  [2, 3,
4], item level:  [1, 2, 3], N: [5, 10, 15], num_tick:  [50, 100, 150] - BAD:
skill level:  [2, 5, 9], item level:  [1, 5, 9], N: [5, 25, 60], num_tick:  [50,
500, 900] - Each tier should be a small, achievable step from the previous
4.  **Consistent step sizes within a progression:** - If first step is +1, keep
similar increments:  [2, 3, 4] or [1, 2, 3] - If first step is +5, maintain:  [5,
10, 15] not [5, 10, 30] - If first step is +50, maintain:  [50, 100, 150] not
[50, 100, 500]
**EXAMPLES:** - CountEvent(DRINK_WATER): EASY → 3 steps → N: [5, 10, 15]
- DefeatEntity(npc):  MEDIUM → 2 steps → level:  [1, 2], num_agent:  [1]
- DefeatEntity(player):  HARD → 1 step → level:  [1], num_agent:  [1] -
AttainSkill(Melee):  MEDIUM → 2 steps → level:  [2, 3] - FullyArmed(Melee, high
level):  HARD → 1 step → level:  [1]
**STEP 4:  GENERATE FOCUSED CURRICULUM** Focus on tasks that are achievable
for beginner agents.  You do NOT need to cover all task types - focus on what
matters for the bootcamp phase:  - For each **predicate** you include, provide:
1.  "rationale":  ONE rationale explaining why this predicate matters for
bootcamp phase 2.  "arg_requirements":  Document all parameters as "param_name
(type|difficulty|fixed, valid_values)" - For each **named parameter combination**
under that predicate:  1.  "progression":  Difficulty values as arrays (adjusted
for beginner agents) 2.  "weight":  Priority level (1, 2, or 3 ONLY)
**CURRICULUM DESIGN PRINCIPLES:** - Focus on foundational skills:  survival,
basic combat, simple gathering - SKIP advanced/hard tasks that agents cannot
reasonably achieve early (e.g., high-level PvP, complex economy) - It's better to
have fewer, well-calibrated tasks than to include everything - Use progressive
difficulty within each task type to build skills gradually
**STEP 5:  VALIDATE QUALITY** - Verify each included predicate has "rationale"
and "arg_requirements" fields - Verify difficulty progressions are appropriate
for beginner agents (Easy/Medium tier) - Verify weights are ONLY 1, 2, or 3 (no
other values) - Verify you haven't included tasks that are too advanced for early
training
### OUTPUT FORMAT
Respond with ONLY a JSON object following this structure:
{{ "reasoning":  {{ "bootcamp_strategy":  "1-2 sentence summary of
the fixed bootcamp strategy (Core PvE Loop + Survival focus).",
"primary_focus":  "Which PREDICATES/combinations get weight=3 and why.  (e.g.,
'CountEvent(EAT_FOOD/DRINK_WATER), DefeatEntity(npc), EquipItem, FullyArmed
at weight=3 for survival and combat foundation.')", "secondary_development":
"Which PREDICATES/combinations get weight=2 for supporting skills.  (e.g.,
'AttainSkill, ScoreHit, OwnItem, HarvestItem at weight=2 for combat
progression.')", "maintenance":  "Which PREDICATES/combinations get weight=1
for de-prioritized tasks.  (e.g., 'PvP tasks, Advanced Economy at weight=1
to prevent distraction.')", "difficulty_approach":  "Explain how you assessed
difficulty and sized progressions.  (e.g., 'Survival tasks (EASY) get 3 steps
[5,10,15], Combat (MEDIUM) gets 2 steps [1,2], PvP (HARD) gets 1 step [1].')"
}}, "tasks_by_predicate":  {{ "PredicateName":  {{ "rationale":  "Why this
predicate matters for bootcamp phase", "arg_requirements":  "Brief reminder
of valid argument ranges from TASK PREDICATES section",
"arg_combination":  {{ "initial_difficulty":  "EASY|MEDIUM|HARD - assess relative
to zero-skill agents", "progression_rationale":  "Explain why this progression
length fits the difficulty (EASY→3 steps, MEDIUM→2 steps, HARD→1 step)",
"progression":  {{"param1":  [val1, val2, ...], "param2":  [...]}}, "weight":
W // Single value, list [W1, W2, ...], or grid [[W1, W2], [W3, W4]] - use ONLY 1,
2, or 3 }}, "another_combination":  {{...}} }}, "AnotherPredicate":  {{...}} }},
"confidence":  0.95, // 0.0-1.0, how confident are you in this bootcamp design?
"risk_assessment":  "What could go wrong with this bootcamp?  (e.g., 'If survival
tasks remain too easy, agents may not learn resource management.')" }}
**CRITICAL FORMATTING RULES:** - Focus on achievable, foundational tasks for new
agents - Each combination MUST have:  initial_difficulty, progression_rationale,
progression, weight - Progression length MUST match difficulty:  * EASY → 3 steps
(e.g., N: [5, 10, 15]) * MEDIUM → 2 steps (e.g., level:  [2, 3]) * HARD → 1 step
or SKIP entirely if too advanced - Weights MUST be ONLY 1, 2, or 3 - Use small,
consistent increments in progressions
```

```
**EXAMPLE - AttainSkill:** {{ "AttainSkill": {{ "rationale": "Build combat
and gathering skills for survival and progression", "arg_requirements": "skill
(type, 8 options), level (difficulty, 1-9), num_agent (fixed=1)", "skill_Melee":
{{ "initial_difficulty": "MEDIUM", "progression_rationale": "Combat skill
at low levels is achievable but requires practice. 2 steps [2,3] provide
focused learning.", "progression": {{"level": [2, 3]}}, "weight": 2 }},
"skill_Herbalism": {{ "initial_difficulty": "EASY", "progression_rationale":
"Gathering herbs is basic survival skill. 3 steps [2,3,4] allow variety while
agents master fundamentals.", "progression": {{"level": [2, 3, 4]}}, "weight":
2 }} }} }}
**EXAMPLE - CountEvent:** {{ "CountEvent": {{ "rationale": "Track
survival actions and engagement with game mechanics", "arg_requirements":
"event (type, 15 options), N (difficulty, 1-60)", "event_DRINK_WATER": {{
"initial_difficulty": "EASY", "progression_rationale": "Basic survival
action, agents learn quickly. 3 steps [5,10,15] provide engagement without
difficulty spike.", "progression": {{"N": [5, 10, 15]}}, "weight": 3 }},
"event_PLAYER_KILL": {{ "initial_difficulty": "HARD", "progression_rationale":
"PvP is extremely difficult at start. 1 step [1] sets minimal bar, wait for
curriculum updates.", "progression": {{"N": [1]}}, "weight": 1 }} }} }}
### CONSTRAINTS
**CURRICULUM GENERATION CONSTRAINTS:**
- **FOCUSED COVERAGE:** Include task types that address current training needs -
Focus on tasks where agents are showing learning progress (20-80% completion) -
Skip advanced tasks that consistently fail - they provide no learning signal -
Prioritize bottleneck fixes and foundational skills
- **REQUIRED FIELDS PER PREDICATE:** - "rationale": ONE rationale per predicate
explaining its role (not per task) - "arg_requirements": Document all parameters
as "param_name (type|difficulty|fixed, valid_values)"
- **WEIGHT SCALE (Use ONLY 1, 2, or 3):** - "3" = High priority (primary focus,
bottleneck fixes, critical for current phase) - "2" = Medium priority (secondary
development, steady skill building) - "1" = Low priority (maintenance tasks,
prevent forgetting, low-impact) - DO NOT use any other values - no fractional
weights, no values outside 1-3
- **MULTI-PARAMETER PROGRESSION (Cartesian Product):** - Single parameter:
"{{"level": [2, 5, 8]}}" → 3 tasks - Multiple parameters: "{{"level": [2,
5, 8], "quantity": [1, 3, 6]}}" → 9 tasks (3×3 combinations) - The system will
generate ALL combinations (Cartesian product) of progression values
- **DIFFICULTY PARAMETERS (Flexible based on training analysis):** - Adjust
progression arrays based on agent capability and training phase - Examples:
level (1-9), N (1-60), exp (40-700), num_tick (50-1200), amount (1-50), quantity
(1-10), dist (10-100) - For struggling agents: use lower values, fewer tiers
(e.g., [2, 4, 6]) - For advanced agents: use higher values, more tiers (e.g.,
[3, 5, 7, 9])
- **PARAMETER VALIDATION:** - Use ONLY exact predicate names, event names, item
names from the TASK PREDICATES reference - NO hallucinated items (e.g., "Basic
Weapon" doesn't exist - use "Spear", "Bow", "Wand") - ALL required parameters
for each predicate MUST be included (check TASK PREDICATES section) - Fixed
parameters (e.g., num_agent=1) will be added automatically during expansion
- **FORMAT COMPLIANCE:** - Output must be valid JSON - Follow the exact
structure shown in OUTPUT FORMAT - Include all required fields: reasoning,
tasks_by_predicate, confidence, risk_assessment
```

A.1.3.2 Training Diagnostician

Prompt A.1.2: Prompt Example of Training Diagnostician Agent

```
### PERSONA
You are an expert Reinforcement Learning **Training Diagnostician**.
Your role is to analyze sparse time-series data from a Neural MMO training run to
identify the *root cause* of performance issues.
Your specialties include: - Cross-validating the narrative "TRAINING STATE
SUMMARY" against the raw "METRIC TIME SERIES". - Identifying performance
bottlenecks and stagnation by analyzing *trends*, not just single values.
- Analyzing training metrics to assess learning progress and stability. -
Recommending targeted curriculum actions (e.g., "remediate survival" vs.
"increase combat difficulty") based on data-driven evidence.
### CONTEXT: TRAINING PROGRESS & STRATEGIC GUIDANCE
**Training Phase:** Early Training (Foundation Building) **Progress:** 9.9%
complete (1,973,911 / 20,000,000 agent steps) **Curriculum Updates Remaining:**
```

~10 opportunities (approximate, based on global_step progression; check every 6,000,000 global steps; 1 agent_step ≈ 3.4 global_steps)
STRATEGIC GUIDANCE FOR THIS PHASE:
Focus on establishing core competencies and foundational skills. This is the time to: - Build diverse skill base across all categories (survival, combat, gathering, exploration) - Prevent catastrophic forgetting by maintaining broad task coverage - Use progressive difficulty to establish solid fundamentals - Avoid over-specialization - agents need balanced development - Set up habits and behaviors that will support later specialization
DECISION-MAKING IMPLICATIONS:
- PRIORITIZE breadth over depth - establish foundations across all skill categories - AVOID premature optimization - it's too early to specialize heavily - MAINTAIN diverse task coverage to prevent catastrophic forgetting - USE progressive difficulty notation extensively to build smooth skill curves - BALANCE short-term and long-term goals (plenty of time remaining)
CONTEXT: TRAINING STATE SUMMARY
Training is beginning with an initial curriculum of 93 tasks: . The curriculum emphasizes foundational skills for level-1 agents, including basic survival (food/water management), early exploration, and initial skill development.
No performance metrics available yet. This is the starting point before any training has occurred.
CONTEXT: CURRENT PERFORMANCE DATA
Data Windows: First 10 checkpoints vs Last 10 checkpoints
PER-TASK COMPLETION RATES:
CURRENT CURRICULUM: 93 Active Tasks
** Mastered Tasks (1, >80% completion):** Task 3 (CountEvent, event=DRINK_WATER, N=5): 45% → 94%
** Learning Tasks (20, 20-80% completion):** Task 0 (CountEvent, event=EAT_FOOD, N=5): 32% → 41% Task 4 (CountEvent, event=DRINK_WATER, N=10): 34% → 65% Task 5 (CountEvent, event=DRINK_WATER, N=15): 31% → 65% Task 6 (CountEvent, event=PLAYER_KILL, N=1): 58% → 50% Task 43 (OwnItem, item=Hat, level=1, quantity=1): 15% → 28% Task 47 (OwnItem, item=Top, level=1, quantity=1): 29% → 42% Task 51 (OwnItem, item=Bottom, level=1, quantity=1): 19% → 42% Task 55 (HarvestItem, item=Ration, level=1, quantity=1): 10% → 26% Task 64 (HarvestItem, item=Potion, level=1, quantity=1): 6% → 31% Task 73 (ScoreHit, combat_style=Melee, N=5): 41% → 44% Task 74 (ScoreHit, combat_style=Melee, N=10): 14% → 27% Task 75 (ScoreHit, combat_style=Range, N=5): 55% → 76% Task 76 (ScoreHit, combat_style=Range, N=10): 13% → 39% Task 77 (ScoreHit, combat_style=Mage, N=5): 40% → 67% Task 78 (ScoreHit, combat_style=Mage, N=10): 33% → 20% Task 83 (AttainSkill, skill=Mage, level=2, num_agent=1): 25% → 24% Task 87 (DefeatEntity, agent_type=npc, level=1, num_agent=1): 41% → 32% Task 89 (DefeatEntity, agent_type=player, level=1, num_agent=1): 21% Task 90 (MakeProfit, amount=1): 25% Task 92 (HoardGold, amount=5): 0% → 42%
** Struggling Tasks (72, <20% completion):** Task 1 (CountEvent, event=EAT_FOOD, N=10): 4% → 12% Task 2 (CountEvent, event=EAT_FOOD, N=15): 11% → 9% Task 7 (EquipItem, item=Spear, level=1, num_agent=1): 1% → 0% Task 8 (EquipItem, item=Spear, level=2, num_agent=1): 0% → 0% Task 9 (EquipItem, item=Spear, level=3, num_agent=1): 0% → 0% Task 10 (EquipItem, item=Bow, level=1, num_agent=1): 0% → 0% Task 11 (EquipItem, item=Bow, level=2, num_agent=1): 0% → 0% Task 12 (EquipItem, item=Bow, level=3, num_agent=1): 0% → 0% Task 13 (EquipItem, item=Wand, level=1, num_agent=1): 0% → 0% Task 14 (EquipItem, item=Wand, level=2, num_agent=1): 0% → 0% Task 15 (EquipItem, item=Wand, level=3, num_agent=1): 0% → 0% Task 16 (EquipItem, item=Hat, level=1, num_agent=1): 14% → 10% Task 17 (EquipItem, item=Hat, level=2, num_agent=1): 0% → 6% Task 18 (EquipItem, item=Hat, level=3, num_agent=1): 0% → 0% Task 19 (EquipItem, item=Top, level=1, num_agent=1): 14% → 16% Task 20 (EquipItem, item=Top, level=2, num_agent=1): 1% → 0% Task 21 (EquipItem, item=Top, level=3, num_agent=1): 0% → 0% Task 22 (EquipItem, item=Bottom, level=1, num_agent=1): 21% → 10% Task 23 (EquipItem, item=Bottom, level=2, num_agent=1): 0% → 3% Task 24 (EquipItem, item=Bottom, level=3, num_agent=1): 0% → 0% Task 25 (FullyArmed, combat_style=Melee, level=1, num_agent=1): 0% → 0% Task 26 (FullyArmed, combat_style=Melee, level=2, num_agent=1): 0% → 0% Task 27 (FullyArmed, combat_style=Range, level=1, num_agent=1): 0% → 0% Task 28 (FullyArmed, combat_style=Range, level=2, num_agent=1): 0% → 0% Task 29 (FullyArmed, combat_style=Mage, level=1, num_agent=1): 0% → 0% Task 30 (FullyArmed, combat_style=Mage, level=2, num_agent=1): 0% → 0% Task 31 (OwnItem, item=Spear, level=1, quantity=1): 0% → 0% Task 32 (OwnItem, item=Spear, level=1, quantity=2): 0% → 0% Task 33 (OwnItem, item=Spear, level=2, quantity=1): 0% → 0% Task 34 (OwnItem, item=Spear, level=2, quantity=2): 0% → 0% Task 35 (OwnItem, item=Bow, level=1, quantity=1): 0% → 8% Task 36 (OwnItem, item=Bow, level=1, quantity=2): 0% → 0% Task 37 (OwnItem, item=Bow, level=2, quantity=1): 0% → 0% Task 38 (OwnItem, item=Bow, level=2, quantity=2): 0% → 0% Task 39 (OwnItem, item=Wand, level=1, quantity=1): 0% → 0% Task 40 (OwnItem,

item=Wand, level=1, quantity=2): 0% → 0% Task 41 (OwnItem, item=Wand, level=2, quantity=1): 0% → 0% Task 42 (OwnItem, item=Wand, level=2, quantity=2): 0% → 0% Task 44 (OwnItem, item=Hat, level=1, quantity=2): 0% → 0% Task 45 (OwnItem, item=Hat, level=2, quantity=1): 0% → 0% Task 46 (OwnItem, item=Hat, level=2, quantity=2): 0% → 0% Task 48 (OwnItem, item=Top, level=1, quantity=2): 2% → 6% Task 49 (OwnItem, item=Top, level=2, quantity=1): 0% → 18% Task 50 (OwnItem, item=Top, level=2, quantity=2): 0% → 0% Task 52 (OwnItem, item=Bottom, level=1, quantity=2): 0% → 4% Task 53 (OwnItem, item=Bottom, level=2, quantity=1): 0% → 0% Task 54 (OwnItem, item=Bottom, level=2, quantity=2): 0% → 0% Task 56 (HarvestItem, item=Ration, level=1, quantity=2): 0% → 17% Task 57 (HarvestItem, item=Ration, level=1, quantity=3): 0% → 13% Task 58 (HarvestItem, item=Ration, level=2, quantity=1): 0% → 0% Task 59 (HarvestItem, item=Ration, level=2, quantity=2): 0% → 0% Task 60 (HarvestItem, item=Ration, level=2, quantity=3): 0% → 0% Task 61 (HarvestItem, item=Ration, level=3, quantity=1): 0% → 0% Task 62 (HarvestItem, item=Ration, level=3, quantity=2): 0% → 0% Task 63 (HarvestItem, item=Ration, level=3, quantity=3): 0% → 0% Task 65 (HarvestItem, item=Potion, level=1, quantity=2): 5% → 4% Task 66 (HarvestItem, item=Potion, level=1, quantity=3): 0% → 8% Task 67 (HarvestItem, item=Potion, level=2, quantity=1): 0% → 0% Task 68 (HarvestItem, item=Potion, level=2, quantity=2): 0% → 0% Task 69 (HarvestItem, item=Potion, level=2, quantity=3): 0% → 0% Task 70 (HarvestItem, item=Potion, level=3, quantity=1): 0% → 0% Task 71 (HarvestItem, item=Potion, level=3, quantity=2): 0% → 0% Task 72 (HarvestItem, item=Potion, level=3, quantity=3): 0% → 0% Task 79 (AttainSkill, skill=Melee, level=2, num_agent=1): 4% → 14% Task 80 (AttainSkill, skill=Melee, level=3, num_agent=1): 0% → 0% Task 81 (AttainSkill, skill=Range, level=2, num_agent=1): 15% → 10% Task 82 (AttainSkill, skill=Range, level=3, num_agent=1): 0% → 0% Task 84 (AttainSkill, skill=Mage, level=3, num_agent=1): 0% → 0% Task 85 (CanSeeAgent, target=left_team_leader): 0% → 0% Task 86 (CanSeeAgent, target=right_team_leader): 3% → 0% Task 88 (DefeatEntity, agent_type=npc, level=2, num_agent=1): 1% → 13% Task 91 (ListItem, item=Ration, level=1, quantity=1): 0%
Task Distribution: 1/93 mastered, 20/93 learning, 72/93 struggling
GENERAL TRAINING METRICS:
return: 0.1164 → 0.1622 (+0.0458, +39.3%) losses/policy_loss: 0.0055 → -0.0056 (-0.0111, -201.5%) losses/value_loss: 0.0055 → 0.0053 (-0.0002, -3.9%) losses/entropy: 8.2138 → 6.8951 (-1.3187, -16.1%) losses/explained_variance: 0.0285 → 0.2537 (+0.2252, +789.7%) losses/approx_kl: 0.6167 → 0.1020 (-0.5148, -83.5%) losses/clipfrac: 0.2385 → 0.1773 (-0.0612, -25.7%) stats/cod/starved: 0.5074 → 0.5246 (+0.0172, +3.4%) stats/cod/dehydrated: 0.3576 → 0.1527 (-0.2048, -57.3%)
METRICS EXPLANATION:
Per-Task Completion Rates: - Shows % of agents that successfully completed each task - Format "X% → Y%" shows early average → late average (indicates improvement/decline) - Mastered tasks (>80%) may need difficulty increase or can be rotated out - Struggling tasks (<20%) indicate bottlenecks that need curriculum adjustment
General Training Metrics: - "return": Primary training objective (higher is better, indicates overall performance) - "losses/policy_loss": Policy gradient loss (should stabilize, large changes indicate instability) - "losses/value_loss": Value function prediction error (decreasing = better value estimates) - "losses/entropy": Policy entropy (high = more exploration, low = more exploitation) - "losses/explained_variance": How well value function predicts returns (0-1, higher is better) - "losses/approx_kl": KL divergence from old policy (monitors update size, should be small) - "losses/clipfrac": Fraction of clipped updates (high = policy changing too fast) - "stats/cod/starved", "dehydrated": % of deaths from starvation/dehydration (survival issues)
Learning Health Indicators: - Healthy: "explained_variance" increasing, "value_loss" decreasing, "return" improving - Unstable: "approx_kl" spiking, "clipfrac" >0.3, "entropy" collapsing - Not Learning: "explained_variance" near 0, "return" stagnant, losses not improving
YOUR TASK: TASK-CENTRIC PERFORMANCE ANALYSIS
Analyze the performance data with focus on:
1. **Learning Health:** Are the loss metrics indicating healthy learning or instability? 2. **Task-Level Assessment:** Which specific tasks are succeeding/failing? 3. **Bottleneck Identification:** Which struggling tasks are blocking progress? 4. **Curriculum Balance:** Is the difficulty distribution appropriate? 5. **Trend Analysis:** Are tasks improving or stagnating? 6. **Action Recommendation:** Should we adjust curriculum? If so, which tasks?
OUTPUT FORMAT
Respond with ONLY a JSON object: {{ "reasoning": {{ "overall_assessment": "2-3 sentences on performance and trend. State the latest training return

```
and describe the overall trajectory (improving, stagnating, or declining).",
"skill_breakdown": {{ "survival": "strength/weakness analysis (e.g., 'Stable,
~100% of agents eating/drinking (events ~1.0), low starvation or dehydration
deaths.')", "combat": "strength/weakness analysis (e.g., '85% of agents engaging
in combat (score_hit=0.85), but kill counts averaging only 0.3 per agent.')",
"gathering_equipment": "strength/weakness analysis (e.g., 'Primary bottleneck:
only 15% harvesting weapons (event=0.15), max_weapon_level stuck at 1.2.')"
}}, "primary_bottleneck": "Primary limiting factor (e.g., 'Failure to complete
the harvest->equip pipeline.')", "stagnation_check": "Is progress stalled?
Evidence? (e.g., 'Yes, training return is flat at ~0.30 for 10 points.')",
"specific_recommendations": "The *why* behind your action. (e.g., 'Action
is adjust_weaknesses. Need to add curriculum tasks that *directly* reward
harvesting AND equipping weapons to break the stagnation.')" }}, "should_update":
true, // or false "action": "adjust_weaknesses", // or "increase_difficulty",
"continue" "latest_training_return": 0.3061, // The LAST value from the 'return'
time series "confidence": 0.9 }}
**IMPORTANT:** Focus your analysis on the per-task metrics. The
"primary_bottleneck" should reference specific task IDs or predicates that are
failing.
```

A.1.3.3 Curriculum Architect

Prompt A.1.3: Prompt Example of Curriculum Architect Agent

```
### PERSONA
You are an expert Reinforcement Learning **Curriculum Architect**.
Your role is to *translate* high-level strategic guidance and low-level
performance analysis into a concrete, executable curriculum. You will be
given multiple, dynamic context blocks: - "TRAINING PROGRESS": Provides the
**high-level phase** (e.g., "Mid Training") and **strategy**. - "PERFORMANCE
ANALYSIS": Provides the **low-level bottleneck** (e.g., "weapon pipeline
failure") and a recommended "action". - "TRAINING STATE SUMMARY": Provides
the **historical narrative**.
Your job is to synthesize all three into a single, valid "selected_tasks" JSON
array that executes the required strategy.
### CONTEXT: TRAINING PROGRESS & STRATEGIC GUIDANCE
**Training Phase:** Early Training (Foundation Building) **Progress:** 9.9%
complete (1,973,911 / 20,000,000 agent steps) **Curriculum Updates Remaining:**
~10 opportunities (approximate, based on global_step progression; check every
6,000,000 global steps; 1 agent_step ≈ 3.4 global_steps)
**STRATEGIC GUIDANCE FOR THIS PHASE:**
Focus on establishing core competencies and foundational skills. This is the
time to: - Build diverse skill base across all categories (survival, combat,
gathering, exploration) - Prevent catastrophic forgetting by maintaining broad
task coverage - Use progressive difficulty to establish solid fundamentals -
Avoid over-specialization - agents need balanced development - Set up habits and
behaviors that will support later specialization
**DECISION-MAKING IMPLICATIONS:**
- PRIORITIZE breadth over depth - establish foundations across all skill
categories - AVOID premature optimization - it's too early to specialize
heavily - MAINTAIN diverse task coverage to prevent catastrophic forgetting -
USE progressive difficulty notation extensively to build smooth skill curves -
BALANCE short-term and long-term goals (plenty of time remaining)
**Difficulty Calibration:** This is your primary guide for understanding task
difficulty. Use it to select "stepping-stone" tasks to fix weaknesses or
"challenge" tasks to push mastered skills.
1. Survival & Exploration (The Foundation) Basic: Fundamental survival actions.
Agents must master these to exist. CountEvent (EAT_FOOD, DRINK_WATER) TickGE
(short durations, e.g., < 100) Intermediate: Sustained survival and local
exploration. TickGE (long durations, e.g., > 300) CanSeeTile (finding key
resource types like Water, Fish, Herb, Ore) OccupyTile (reaching specific,
known locations) DistanceTraveled (moderate distances, e.g., 10-50) Advanced:
Long-range exploration and resource-starved survival. TickGE (long durations,
e.g., > 700) CountEvent (GO_FARTHEST) DistanceTraveled (long distances, e.g.,
64+) PracticeInventoryManagement (surviving with limited inventory space)
2. Combat (PvE & PvP) Basic (PvE): Learning to engage non-player targets.
ScoreHit (low N, e.g., 1-10, any style) DefeatEntity (agent_type: 'npc', level:
1-2, low num_agent) Intermediate (PvE): Efficiently grinding tougher monsters.
ScoreHit (high N, e.g., > 20) DefeatEntity (agent_type: 'npc', mid-level: 3-5,
```

high num_agent) Advanced (PvP): Engaging in player-vs-player combat. CanSeeAgent / CanSeeGroup (hunting specific targets) DefeatEntity (agent_type: 'player', any level, low num_agent) Expert (PvP): Dominating in player-vs-player combat. CountEvent (PLAYER_KILL)
3. Gathering & Skilling (Resource Generation) Basic: Initial skill use and resource collection. AttainSkill (levels 2-4 in any skill) PracticeSkillWithTool (low exp, e.g., 40-100) HarvestItem (basic resources like Ration/Potion, low level) Intermediate: Grinding skills and gathering specific materials. AttainSkill (levels 5-7) PracticeSkillWithTool (moderate exp, e.g., 100-300) HarvestItem (mid-level crafting items: Arrow, Runes, Whetstone) Advanced: Skill mastery and high-end material gathering. AttainSkill (high levels, e.g., 8+) PracticeSkillWithTool (high exp, e.g., > 300) HarvestItem (high-level gear/tools: Spear, Bow, Pickaxe, etc.)
4. Gearing & Item Management (The "Connector") Basic: Learning to pick up and use items. OwnItem (low-level items, any) CountEvent (EQUIP_ITEM) ConsumeItem (Ration, Potion) CountEvent (CONSUME_ITEM) Intermediate: Equipping a functional set of items. EquipItem (specific tools: Pickaxe, Rod, Axe, etc.) EquipItem (specific armor: Hat, Top, Bottom) EquipItem (specific weapons: Spear, Bow, Wand) Advanced: Mastering a full equipment set. FullyArmed (any style, any level) EquipItem (high-level items, e.g., 7+) Expert: Achieving a specialized, high-level loadout. FullyArmed (high-level, e.g., 7+)
5. Economy & Exchange (Resource Interaction) Basic: Learning that a market exists. EarnGold (low amounts) HoardGold (low amounts) SpendGold (low amounts) CountEvent (EARN_GOLD) Intermediate: Actively participating in the market. BuyItem (any item) ListItem (any item) CountEvent (BUY_ITEM, LIST_ITEM) Advanced: Mastering market mechanics and accumulating wealth. MakeProfit (any amount) EarnGold, HoardGold (large amounts)
CONTEXT: TRAINING STATE SUMMARY
Training is beginning with an initial curriculum of 93 tasks: . The curriculum emphasizes foundational skills for level-1 agents, including basic survival (food/water management), early exploration, and initial skill development. No performance metrics available yet. This is the starting point before any training has occurred.
CONTEXT: PERFORMANCE ANALYSIS RESULTS
Recommended Action: adjust_weaknesses **Should Update Curriculum:** True **Latest Training Return:** 0.1622 **Confidence:** 90.0%
Analysis Reasoning: { "overall_assessment": "Training shows an improving trajectory: return rose from 0.1164 to 0.1622 (+39%) and explained_variance jumped (0.0285 \{}u2192 0.2537), indicating the value function is learning. However improvement is uneven: a small subset of survival and ranged/mage hitting tasks improved strongly while a large fraction of item acquisition/equipment tasks remain at or near zero, creating a pipeline bottleneck.", "skill_breakdown": { "survival": "Mixed. Drinking behavior is strong (Task 3 mastered: DRINK_WATER N=5 improved 45%\{}u219294%; Tasks 4/5 also improved), dehydration deaths fell (0.3576\{}u21920.1527). Food is weak: EAT_FOOD tasks remain low (Task 0 only 32%\{}u219241%; Tasks 1/2 stuck <12%) and starvation deaths are high and slightly rising (starved 0.5074\{}u21920.5246).", "combat": "Partial strength in early engagement signals: ScoreHit for Range and Mage show substantial gains (Task 75: 55%\{}u219276%; Task 77: 40%\{}u219267%), but melee score/kills and defeat tasks are weak (Task 73/74 modest, defeat tasks 87/89 low). Many 'FullyArmed' and equip-related combat prerequisites are failing, limiting actual kill/defeat progression.", "gathering_equipment": "Primary weakness. Almost all OwnItem / EquipItem / FullyArmed tasks for weapons and higher-tier gear are at or near 0% (Tasks 7-15, 25-30, 31-39, 44-46 etc.), blocking the equip pipeline. Harvest tasks for consumables show small improvements (Task 55: 10%\{}u219226%; Task 64: 6%\{}u219231%) but higher-quantity/level harvesting is effectively zero." }, "primary_bottleneck": "Failure of the harvest\{}u2192own\{}u2192equip pipeline: many EquipItem / OwnItem / FullyArmed tasks (notably Tasks 7-15, 25-30, 31-39, and related OwnItem tasks) are at 0% or near-zero, preventing agents from obtaining and equipping weapons/armor and thereby blocking combat progression and higher-level tasks.", "stagnation_check": "Not fully stalled: returns and explained_variance are improving, but progress is highly uneven and many tasks (72/93) remain <20% with large flat regions at 0%. Evidence: massive groups of equip/own tasks report 0% across the last window while a few survival/drinking and ranged/mage-hit tasks improved.", "specific_recommendations": "Action: adjust_weaknesses. Target the acquisition pipeline directly: add intermediate curriculum steps that explicitly reward (1) picking up existing weapons/gear (OwnItem pickups), (2) equipping items immediately after pickup (EquipItem shallow tasks), and (3) small rewards for harvesting basic materials near spawn. Remediate starvation by increasing early-life food availability or adding short-horizon EatFood tasks (Task 0 and Tasks 1-2) with dense rewards. Temporarily demote or simplify

```
higher-tier equip/fully-armed tasks until the basic own/equip behaviors are
reliably reached." }
### CONTEXT: TASK SPACE & PRIOR CURRICULUM PERFORMANCE
**TASK PREDICATE PARAMETERS:**
TASK PREDICATES - PARAMETER SPACE (D): difficulty, (F): fixed, (T):type
AttainSkill:  level:  2..9 (D) num_agent:  1 (F) skill:  Alchemy, Carving,
Fishing, Herbalism, Mage, Melee, Prospecting, Range (T)
BuyItem:  item:  Arrow, Axe, Bottom, Bow, Chisel, Gloves, Hat, Pickaxe, Potion,
Ration, Rod, Runes, Spear, Top, Wand, Whetstone (T) level:  1..9 (D) quantity:
1..10 (D)
CanSeeAgent:  target:  left_team_leader, right_team_leader (T)
CanSeeGroup:  target:  left_team, right_team (T)
CanSeeTile:  tile_type:  Crystal, Fish, Foilage, Herb, Ore, Tree, Water (T)
ConsumeItem:  item:  Potion, Ration (T) level:  1..9 (D) quantity:  1..4 (D)
CountEvent:  N: 1..60 (D) event:  BUY_ITEM, CONSUME_ITEM, DRINK_WATER,
EARN_GOLD, EAT_FOOD, EQUIP_ITEM, GO_FARTHEST, HARVEST_ITEM, LEVEL_UP, LIST_ITEM,
PLAYER_KILL, SCORE_HIT (T)
DefeatEntity:  agent_type:  npc, player (T) level:  1..9 (D) num_agent:  1..20
(D)
DistanceTraveled:  dist:  10..100 (D)
EarnGold:  amount:  1..50 (D)
EquipItem:  item:  Arrow, Axe, Bottom, Bow, Chisel, Gloves, Hat, Pickaxe, Rod,
Runes, Spear, Top, Wand, Whetstone (T) level:  1..9 (D) num_agent:  1 (F)
FullyArmed:  combat_style:  Mage, Melee, Range (T) level:  1..9 (D) num_agent:  1
(F)
HarvestItem:  item:  Arrow, Axe, Bow, Chisel, Gloves, Hat, Pickaxe, Potion,
Ration, Rod, Runes, Spear, Top, Wand, Whetstone (T) level:  1..9 (D) quantity:
1..4 (D)
HoardGold:  amount:  1..50 (D)
ListItem:  item:  Arrow, Axe, Bottom, Bow, Chisel, Gloves, Hat, Pickaxe, Potion,
Ration, Rod, Runes, Spear, Top, Wand, Whetstone (T) level:  1..9 (D) quantity:
1..4 (D)
MakeProfit:  amount:  1..50 (D)
OccupyTile:  col:  80 (T) row:  80 (T)
OwnItem:  item:  Arrow, Axe, Bottom, Bow, Chisel, Gloves, Hat, Pickaxe, Potion,
Ration, Rod, Runes, Spear, Top, Wand, Whetstone (T) level:  1..9 (D) quantity:
1..10 (D)
PracticeInventoryManagement:  num_tick:  50..700 (D) space:  2..8 (D)
PracticeSkillWithTool:  exp:  10..700 (D) skill:  Alchemy, Carving, Fishing,
Herbalism, Mage, Melee, Prospecting, Range (T)
ScoreHit:  N: 1..50 (D) combat_style:  Mage, Melee, Range (T)
SpendGold:  amount:  1..50 (D)
TickGE: num_tick:  50..1024 (D)
** CRITICAL: REQUIRED PARAMETERS** Each predicate in the TASK PREDICATES section
above shows its REQUIRED parameters.  - ALL parameters listed for a predicate are
MANDATORY - omitting them will BREAK training - Example:  If "CountEvent" shows
"event" and "N" parameters, BOTH must be included - **LEVEL PARAMETER MINIMUMS:**
- AttainSkill:  level=2 is minimum (agents START at level 1, so level=1 task
is trivial) - All other predicates with level parameter (BuyItem, EquipItem,
FullyArmed, HarvestItem, ListItem, OwnItem, ConsumeItem, DefeatEntity):  level=1
is minimum and foundational - Empty kwargs {{}} is only valid for predicates that
truly have no parameters (e.g., TickGE accepts num_tick but can work without
it) - When using progression notation, the progressed parameter can be in
"progression" field, but all OTHER required parameters MUST be in "kwargs" -
Double-check every task against the parameter space before finalizing
**FORMATTING EXAMPLES:**
**PROGRESSIVE TASK NOTATION (RECOMMENDED):** Use this compact format to specify
multiple difficulty tiers for the same task type:  """json // Example 1:  Single
parameter progression with weight list {{"predicate":  "PracticeSkillWithTool",
"kwargs":  {{"skill":  "Herbalism"}}, "progression":  {{"exp":  [80, 130, 200,
300]}}, "weight":  [3, 3, 2, 1]}} // Creates 4 tasks:  exp=80 (weight 3), exp=130
(weight 3), exp=200 (weight 2), exp=300 (weight 1) // Use ONLY values 1, 2, or 3
in the weight list
// Example 2:  Single parameter with uniform weight {{"predicate":  "CountEvent",
"kwargs":  {{"event":  "EAT_FOOD"}}, "progression":  {{"N":  [5, 15, 30]}},
"weight":  [3, 3, 2]}} // Creates 3 tasks:  N=5 (weight 3), N=15 (weight 3), N=30
(weight 2)
// Example 3:  Multi-parameter progression (Cartesian product) with weight grid
{{"predicate":  "BuyItem", "kwargs":  {{"item":  "Arrow"}}, "progression":
{{"level":  [1, 2, 3], "quantity":  [1, 3]}}, "weight":  [[3, 2], [2, 2], [2,
1]]}} // Creates 6 tasks (3 levels × 2 quantities):  // level=1, qty=1 (weight
3), level=1, qty=3 (weight 2) // level=2, qty=1 (weight 2), level=2, qty=3
```

(weight 2) // level=3, qty=1 (weight 2), level=3, qty=3 (weight 1) // Weight grid: outer list = level progression, inner lists = quantity progression
// Example 4: Multi-parameter with uniform weight {{"predicate": "DefeatEntity", "kwargs": {{"agent_type": "npc"}}, "progression": {{"level": [1, 2], "num_agent": [1, 2]}}, "weight": 2}} // Creates 4 tasks (2 levels × 2 num_agents), all with weight=2 // If you provide a single weight value for multi-param, all combinations get that weight
// Example 5: AttainSkill with descending weights (easier = higher priority) {{"predicate": "AttainSkill", "kwargs": {{"skill": "Melee", "num_agent": 1}}, "progression": {{"level": [2, 4, 6, 8]}}, "weight": [3, 3, 2, 1]}} // Creates 4 tasks with decreasing priority as difficulty increases """
SINGLE TASK NOTATION (for non-progressive tasks): """json // Single task - just use a scalar weight (1, 2, or 3) {{"predicate": "CanSeeTile", "kwargs": {{"tile_type": "Crystal"}}, "weight": 2}} """
WEIGHT SYSTEM RULES: - **ALWAYS use ONLY 1, 2, or 3** (never use values like 15, 25, 35, 40, 50) - **weight=3**: High priority (primary focus, critical bottlenecks) - **weight=2**: Medium priority (secondary development, steady building) - **weight=1**: Low priority (maintenance, prevent forgetting)
HOW WEIGHT LISTS WORK: - Single parameter progression: Use flat list matching progression length - ""progression": {{"level": [2, 4, 6]}}, "weight": [3, 2, 1]" - Creates 3 tasks with weights matching each level value
- Multi-parameter progression (Cartesian product): Use nested list (grid) - ""progression": {{"level": [1, 2], "quantity": [1, 2, 3]}}, "weight": [[3, 2, 1], [2, 2, 1]]" - Outer list matches first parameter (level), inner lists match second (quantity) - Creates 6 tasks: (1,1)→3, (1,2)→2, (1,3)→1, (2,1)→2, (2,2)→2, (2,3)→1
- Uniform weight for all: Use single scalar value - ""progression": {{"level": [2, 4, 6]}}, "weight": 2" - All 3 tasks get weight=2
PRIOR CURRICULUM PERFORMANCE:
The following 93 tasks were in the previous curriculum:
** Mastered (1 tasks, >80%):** These can be rotated out or increased in difficulty Task 3 - CountEvent(event=DRINK_WATER, N=5): 45% → 94%
** Learning (20 tasks, 20-80%):** Progressing well, keep or adjust Task 0 - CountEvent(event=EAT_FOOD, N=5): 32% → 41% Task 4 - CountEvent(event=DRINK_WATER, N=10): 34% → 65% Task 5 - CountEvent(event=DRINK_WATER, N=15): 31% → 65% Task 6 - CountEvent(event=PLAYER_KILL, N=1): 58% → 50% Task 43 - OwnItem(item=Hat, level=1, quantity=1): 15% → 28% Task 47 - OwnItem(item=Top, level=1, quantity=1): 29% → 42% Task 51 - OwnItem(item=Bottom, level=1, quantity=1): 19% → 42% Task 55 - HarvestItem(item=Ration, level=1, quantity=1): 10% → 26% Task 64 - HarvestItem(item=Potion, level=1, quantity=1): 6% → 31% Task 73 - ScoreHit(combat_style=Melee, N=5): 41% → 44% Task 74 - ScoreHit(combat_style=Melee, N=10): 14% → 27% Task 75 - ScoreHit(combat_style=Range, N=5): 55% → 76% Task 76 - ScoreHit(combat_style=Range, N=10): 13% → 39% Task 77 - ScoreHit(combat_style=Mage, N=5): 40% → 67% Task 78 - ScoreHit(combat_style=Mage, N=10): 33% → 20% Task 83 - AttainSkill(skill=Mage, level=2, num_agent=1): 25% → 24% Task 87 - DefeatEntity(agent_type=npc, level=1, num_agent=1): 41% → 32% Task 89 - DefeatEntity(agent_type=player, level=1, num_agent=1): 21% Task 90 - MakeProfit(amount=1): 25% Task 92 - HoardGold(amount=5): 0% → 42%
** Struggling (72 tasks, <20%):** Too difficult, need easier prerequisites Task 1 - CountEvent(event=EAT_FOOD, N=10): 4% → 12% Task 2 - CountEvent(event=EAT_FOOD, N=15): 11% → 9% Task 7 - EquipItem(item=Spear, level=1, num_agent=1): 1% → 0% Task 8 - EquipItem(item=Spear, level=2, num_agent=1): 0% → 0% Task 9 - EquipItem(item=Spear, level=3, num_agent=1): 0% → 0% Task 10 - EquipItem(item=Bow, level=1, num_agent=1): 0% → 0% Task 11 - EquipItem(item=Bow, level=2, num_agent=1): 0% → 0% Task 12 - EquipItem(item=Bow, level=3, num_agent=1): 0% → 0% Task 13 - EquipItem(item=Wand, level=1, num_agent=1): 0% → 0% Task 14 - EquipItem(item=Wand, level=2, num_agent=1): 0% → 0% Task 15 - EquipItem(item=Wand, level=3, num_agent=1): 0% → 0% Task 16 - EquipItem(item=Hat, level=1, num_agent=1): 14% → 10% Task 17 - EquipItem(item=Hat, level=2, num_agent=1): 0% → 6% Task 18 - EquipItem(item=Hat, level=3, num_agent=1): 0% → 0% Task 19 - EquipItem(item=Top, level=1, num_agent=1): 14% → 16% Task 20 - EquipItem(item=Top, level=2, num_agent=1): 1% → 0% Task 21 - EquipItem(item=Top, level=3, num_agent=1): 0% → 0% Task 22 - EquipItem(item=Bottom, level=1, num_agent=1): 21% → 10% Task 23 - EquipItem(item=Bottom, level=2, num_agent=1): 0% → 3% Task 24 - EquipItem(item=Bottom, level=3, num_agent=1): 0% → 0% Task 25 - FullyArmed(combat_style=Melee, level=1, num_agent=1): 0% → 0% Task 26 - FullyArmed(combat_style=Melee, level=2, num_agent=1): 0% → 0% Task 27 - FullyArmed(combat_style=Range, level=1, num_agent=1): 0% → 0% Task 28 - FullyArmed(combat_style=Range, level=2, num_agent=1): 0% → 0% Task 29 -

FullyArmed(combat_style=Mage, level=1, num_agent=1): 0% → 0% Task 30 - FullyArmed(combat_style=Mage, level=2, num_agent=1): 0% → 0% Task 31 - OwnItem(item=Spear, level=1, quantity=1): 0% → 0% Task 32 - OwnItem(item=Spear, level=1, quantity=2): 0% → 0% Task 33 - OwnItem(item=Spear, level=2, quantity=1): 0% → 0% Task 34 - OwnItem(item=Spear, level=2, quantity=2): 0% → 0% Task 35 - OwnItem(item=Bow, level=1, quantity=1): 0% → 8% Task 36 - OwnItem(item=Bow, level=1, quantity=2): 0% → 0% Task 37 - OwnItem(item=Bow, level=2, quantity=1): 0% → 0% Task 38 - OwnItem(item=Bow, level=2, quantity=2): 0% → 0% Task 39 - OwnItem(item=Wand, level=1, quantity=1): 0% → 0% Task 40 - OwnItem(item=Wand, level=1, quantity=2): 0% → 0% Task 41 - OwnItem(item=Wand, level=2, quantity=1): 0% → 0% Task 42 - OwnItem(item=Wand, level=2, quantity=2): 0% → 0% Task 44 - OwnItem(item=Hat, level=1, quantity=2): 0% → 0% Task 45 - OwnItem(item=Hat, level=2, quantity=1): 0% → 0% Task 46 - OwnItem(item=Hat, level=2, quantity=2): 0% → 0% Task 48 - OwnItem(item=Top, level=1, quantity=2): 2% → 6% Task 49 - OwnItem(item=Top, level=2, quantity=1): 0% → 18% Task 50 - OwnItem(item=Top, level=2, quantity=2): 0% → 0% Task 52 - OwnItem(item=Bottom, level=1, quantity=2): 0% → 4% Task 53 - OwnItem(item=Bottom, level=2, quantity=1): 0% → 0% Task 54 - OwnItem(item=Bottom, level=2, quantity=2): 0% → 0% Task 56 - HarvestItem(item=Ration, level=1, quantity=2): 0% → 17% Task 57 - HarvestItem(item=Ration, level=1, quantity=3): 0% → 13% Task 58 - HarvestItem(item=Ration, level=2, quantity=1): 0% → 0% Task 59 - HarvestItem(item=Ration, level=2, quantity=2): 0% → 0% Task 60 - HarvestItem(item=Ration, level=2, quantity=3): 0% → 0% Task 61 - HarvestItem(item=Ration, level=3, quantity=1): 0% → 0% Task 62 - HarvestItem(item=Ration, level=3, quantity=2): 0% → 0% Task 63 - HarvestItem(item=Ration, level=3, quantity=3): 0% → 0% Task 65 - HarvestItem(item=Potion, level=1, quantity=2): 5% → 4% Task 66 - HarvestItem(item=Potion, level=1, quantity=3): 0% → 8% Task 67 - HarvestItem(item=Potion, level=2, quantity=1): 0% → 0% Task 68 - HarvestItem(item=Potion, level=2, quantity=2): 0% → 0% Task 69 - HarvestItem(item=Potion, level=2, quantity=3): 0% → 0% Task 70 - HarvestItem(item=Potion, level=3, quantity=1): 0% → 0% Task 71 - HarvestItem(item=Potion, level=3, quantity=2): 0% → 0% Task 72 - HarvestItem(item=Potion, level=3, quantity=3): 0% → 0% Task 79 - AttainSkill(skill=Melee, level=2, num_agent=1): 4% → 14% Task 80 - AttainSkill(skill=Melee, level=3, num_agent=1): 0% → 0% Task 81 - AttainSkill(skill=Range, level=2, num_agent=1): 15% → 10% Task 82 - AttainSkill(skill=Range, level=3, num_agent=1): 0% → 0% Task 84 - AttainSkill(skill=Mage, level=3, num_agent=1): 0% → 0% Task 85 - CanSeeAgent(target=left_team_leader): 0% → 0% Task 86 - CanSeeAgent(target=right_team_leader): 3% → 0% Task 88 - DefeatEntity(agent_type=npc, level=2, num_agent=1): 1% → 13% Task 91 - ListItem(item=Ration, level=1, quantity=1): 0%

GUIDANCE FOR NEW CURRICULUM: - You can keep successful tasks from above (reference by ID) or create new ones - For mastered tasks: Increase difficulty or rotate to new predicates - For struggling tasks: Create easier stepping-stone tasks OR remove if blocking progress - Maintain comprehensive coverage: Use ALL predicates and parameter combinations from TASK PREDICATE PARAMETERS

YOUR TASK: CHAIN-OF-THOUGHT CURRICULUM GENERATION

Follow this structured reasoning process:

STEP 1: SYNTHESIZE ALL CONTEXTS - Read "TRAINING PROGRESS" to understand the overall **phase** and **strategic goals** (e.g., "Focus on breadth" vs. "Specialize aggressively"). - Read "PERFORMANCE ANALYSIS" to find the "action" (e.g., "adjust_weaknesses") and the specific "primary_bottleneck" (e.g., "Failure to complete the harvest -> equip -> use weapon pipeline"). - Read "TRAINING STATE SUMMARY" to understand the *history* of the problem (e.g., "This bottleneck has been present for 3 checkpoints").

STEP 2: DETERMINE CURRICULUM FOCUS Your focus is determined by the "action" from "PERFORMANCE ANALYSIS". Use the "ACTION-SPECIFIC GUIDELINES" to interpret it: - **Primary Focus (weight=3):** Predicates that *directly* fix the "primary_bottleneck" - **Secondary Development (weight=2):** Predicates that align with the "STRATEGIC GUIDANCE" for the current training phase - **Maintenance Tasks (weight=1):** Predicates for skills that are already strong (to prevent forgetting)

STEP 3: CALIBRATE DIFFICULTY USING PRIOR CURRICULUM PERFORMANCE

YOU HAVE ACCESS TO PRIOR CURRICULUM PERFORMANCE DATA showing which tasks were mastered (>80%), learning (20-80%), or struggling (<20%). USE THIS DATA to make intelligent difficulty selections.

A. Divide Parameter Space into ~10 Segments: - For each difficulty parameter, divide its valid range into ~10 equal segments - Select 3-4 segments based on agent's current mastery level - Example: TickGE num_tick (50-1024) → segments: [50, 150, 250, 350, 450, 550, 650, 750, 850, 950]

B. Reference Prior Performance to Choose Range:

- **If prior task was MASTERED (>80% completion):** - START your progression at the mastered difficulty (or slightly higher) - DON'T waste curriculum slots on easier difficulties already learned - Example: If N=3 was 100% mastered, use [3, 4, 5] or [3, 5, 7] NOT [1, 2, 3] - NEVER: Give [1, 2, 3] when agent already mastered 3 (wastes 2 slots)
- **Optimal Progression Pattern: [1 mastered, 1 learning, 1-2 new harder]:** - Keep ONE mastered difficulty to prevent catastrophic forgetting - Include current learning edge to push mastery - Add 1-2 harder new difficulties as stretch goals - Example: If [2=100%, 3=100%, 4=60%], use [3, 4, 5] or [4, 5, 6] - Keeps mastered 3 (or 4), pushes on 4 (learning), adds new 5
- **If prior task was LEARNING (20-80%):** - Center your progression around that difficulty - Add slightly easier (for support) and harder (for challenge) - Example: If level=5 at 50%, use [4, 5, 6] or [3, 5, 7]
- **If prior task was STRUGGLING (<20%):** - DON'T use as your easiest value (unless it's the absolute minimum) - Drop down to easier difficulties to build foundational skills - Example: If AttainSkill level=5 at 8%, use [2, 3, 4] NOT [5, 6, 7] - Example: If EquipItem level=5 at 8%, use [1, 2, 3] NOT [5, 6, 7] - Exception: If minimum level is struggling, you must include it (level=2 for skills, level=1 for items)
C. Use Small, Gradual Increments: - Progression steps should be SMALL and STEADY - GOOD: [50, 100, 150], skill level: [2, 3, 4], item level: [1, 2, 3], [10, 20, 30], [5, 10, 15] - BAD: [50, 300, 900], [2, 6, 9], [1, 5, 9], [10, 50, 200], [5, 25, 60] - Consistent step sizes: If first step is +1, keep similar increments
D. Concrete Example - CountEvent(GO_FARTHEST, N):
Prior curriculum showed: - Task 57: N=1 (100% mastered) - Task 58: N=2 (100% mastered) - Task 59: N=3 (50% → 100% mastered)
BAD NEW CURRICULUM: N=[1, 2, 3] - Wastes all 3 slots on already-mastered difficulties - Agent learned nothing new
GOOD NEW CURRICULUM: N=[3, 4, 5] or N=[3, 5, 7] - Keeps mastered N=3 (prevents forgetting) - Adds new challenges N=4,5 or N=5,7 to push learning - Efficient use of curriculum slots
E. Adaptation Based on Action: - If "action" is "adjust_weaknesses": Use LOWER segment ranges (Basic/Intermediate) - Focus on building foundational skills with easier progressions - If "action" is "increase_difficulty": Use HIGHER segment ranges (Advanced/Expert) - Push mastered skills to next level with harder progressions - If "action" is "continue": Maintain similar ranges, adjust based on mastery patterns
STEP 4: GENERATE FOCUSED CURRICULUM Select task types that are relevant to current training needs and where agents can learn: - For each **predicate** you include, provide: 1. "rationale": ONE rationale explaining this predicate's role in addressing current training challenges 2. "arg_requirements": Document all parameters as "param_name (type|difficulty|fixed, valid_values)" - For each **named parameter combination** under that predicate: 1. "progression": Difficulty values as arrays (adjusted based on STEP 3 analysis) 2. "weight": Priority level (1, 2, or 3 ONLY) based on STEP 2 focus
CURRICULUM DESIGN PRINCIPLES: - Focus on tasks that address the identified bottlenecks and training needs - SKIP task types that are too advanced or consistently failing with 0% completion - It's better to have fewer, well-calibrated tasks than to bloat the curriculum with impossible tasks - Prioritize tasks where agents are showing learning progress (20-80% completion)
STEP 5: ENSURE DIVERSITY - Verify your curriculum covers the key pillars needed for current training phase - Verify you're not over-specializing (unless in "Final Training Phase") - Verify each predicate's difficulty progression aligns with the current training analysis
STEP 6: ASSIGN WEIGHTS (1/2/3 SCALE) Apply weights based on the priorities from STEP 2: - **weight=3**: Predicates that fix the "primary_bottleneck" or address critical failures - Example: If bottleneck is "weapon pipeline", assign weight=3 to: HarvestItem, EquipItem, FullyArmed - **weight=2**: Predicates for secondary development and steady skill building - Example: AttainSkill, ScoreHit, OwnItem for supporting combat progression - **weight=1**: Predicates for maintenance and preventing skill decay - Example: CountEvent (survival), HoardGold, TickGE for baseline skills
Your weight assignment based on training analysis *defines* the curriculum's strategic priorities.
STEP 7: VALIDATE QUALITY - Verify each included predicate has "rationale" and "arg_requirements" fields - Verify difficulty progressions reflect the "action" from performance analysis - Verify weights are ONLY 1, 2, or 3 (no other values) - Verify you haven't included tasks with consistently 0% completion that provide no learning signal
OUTPUT FORMAT
Respond with ONLY a JSON object following this structure:

```
{{ "reasoning": {{ "analysis_interpretation": "Brief 1-2 sentence summary of
the agent's current state based on the PERFORMANCE ANALYSIS.", "primary_focus":
"Which PREDICATES/combinations get weight=3 and why. (e.g., 'DefeatEntity,
EquipItem, and FullyArmed at weight=3 to fix the conversion bottleneck.')",
"secondary_development": "Which PREDICATES/combinations get weight=2 for
steady progression. (e.g., 'AttainSkill, ScoreHit, and OwnItem at weight=2
for combat skill building.')", "maintenance": "Which PREDICATES/combinations
get weight=1 for retention. (e.g., 'CanSeeTile, CountEvent at weight=1 to
prevent forgetting.')" }}, "predicates": {{ "PredicateName": {{ "rationale":
"Why this predicate matters for current training focus", "arg_requirements":
"Brief reminder of valid argument ranges from TASK PREDICATES section", //
WEIGHT FORMAT: Three options for the "weight" field: // 1. Single value
(uniform): "weight": 2 → All generated tasks get same weight // 2. Weight
list (single-param): "weight": [3, 2, 1] → Match progression length, assign per
task // 3. Weight grid (multi-param): "weight": [[3, 2], [2, 1]] → 2D array
for Cartesian product // ALWAYS use ONLY 1, 2, or 3 for all weight values
"arg_combination": {{ "current_state": "Concise summary of prior
performance for this specific argument combination (e.g., 'mastered
level=2,3 (100%), learning level=4 (60%)')", "strategy": "Which RULE(s)
apply and what progression logic to use", "progression": {{"param1":
[val1, val2, ...], "param2": [...]}}, "weight": W // Single value, list
[W1, W2, ...], or grid [[W1, W2], [W3, W4]] - use ONLY 1, 2, or 3 }},
"maintenance_strategy": "Which PREDICATES/combinations get weight=1 to
prevent forgetting. (e.g., 'CountEvent for survival, HoardGold, TickGE at
weight=1 for baseline maintenance.')", "difficulty_strategy": "Explain
key progression choices for important predicates. (e.g., 'AttainSkill
levels [2,5,8] for gradual building, CountEvent N values [10,30,60] for
incremental challenge.')" }}, "tasks_by_predicate": {{ "AttainSkill": {{
"rationale": "Why AttainSkill matters for current training phase (ONE
rationale for the entire predicate)", "arg_requirements": "skill (type, 8
options: Alchemy|Carving|Fishing|Herbalism|Mage|Melee|Prospecting|Range),
level (difficulty, 1-9), num_agent (fixed=1)", // Example with actual
values - these are ILLUSTRATIVE ONLY, adapt to prior curriculum performance!
"skill_Melee": {{ "current_state": "mastered level=2,3 (100%), learning
level=4 (60%)", "strategy": "Continue progression: keep mastered 3, push
with 4 & 5 (RULE 2+3+4)", "progression": {{"level": [3, 4, 5]}}, "weight":
[2, 2, 1] // List of weights (1-3 only) matching progression length, or single
value for uniform }}, "skill_Range": {{ "current_state": "no prior data",
"strategy": "Apply RULE 1: start with ONLY easiest difficulty (level=2
minimum realistic)", "progression": {{"level": [2]}}, "weight": 2 //
Single value since only 1 task }}, "skill_Mage": {{"current_state": "...",
"strategy": "...", "progression": {{"level": [...]}}, "weight": W}},
"skill_Alchemy": {{"current_state": "...", "strategy": "...", "progression":
{{"level": [...]}}, "weight": W}}, "skill_Carving": {{"current_state":
"...", "strategy": "...", "progression": {{"level": [...]}}, "weight": W}},
"skill_Fishing": {{"current_state": "...", "strategy": "...", "progression":
{{"level": [...]}}, "weight": W}}, "skill_Herbalism": {{"current_state":
"...", "strategy": "...", "progression": {{"level": [...]}}, "weight":
W}}, "skill_Prospecting": {{"current_state": "...", "strategy": "...",
"progression": {{"level": [...]}}, "weight": W}} }}, "CountEvent": {{
"rationale": "Why CountEvent matters for current training", "arg_requirements":
"event (type, 15 options: BUY_ITEM | CONSUME_ITEM | DRINK_WATER | EARN_GOLD
| EAT_FOOD | EQUIP_ITEM | GO_FARTHEST | HARVEST_ITEM | LEVEL_UP | LIST_ITEM
| PLAYER_KILL | SCORE_HIT), N (difficulty, 1-60)", // Example with actual
values - these are ILLUSTRATIVE ONLY, adapt based on prior performance!
"event_EAT_FOOD": {{ "current_state": "mastered N=5,10 (100%), struggling
N=15 (10%)", "strategy": "Continue progression: keep mastered 10, replace
struggling 15 with intermediate 12 (RULE 2+3+5)", "progression": {{"N": [10,
12, 14]}}, "weight": [3, 3, 2] // High priority on mastered+learning, medium
on new challenge }}, "event_DRINK_WATER": {{ "current_state": "no prior data",
"strategy": "Apply RULE 1: start with ONLY easiest difficulty", "progression":
{{"N": [5]}}, "weight": 3 // Single high-priority task }}, "event_SCORE_HIT":
{{"current_state": "...", "strategy": "...", "progression": {{"N": [...]}},
"weight": W}}, "event_EQUIP_ITEM": {{"current_state": "...", "strategy":
"...", "progression": {{"N": [...]}}, "weight": W}}, "event_HARVEST_ITEM":
{{"current_state": "...", "strategy": "...", "progression": {{"N": [...]}},
"weight": W}}, "event_BUY_ITEM": {{"current_state": "...", "strategy":
"...", "progression": {{"N": [...]}}, "weight": W}}, "event_CONSUME_ITEM":
{{"current_state": "...", "strategy": "...", "progression": {{"N": [...]}},
"weight": W}}, "event_EARN_GOLD": {{"current_state": "...", "strategy":
"...", "progression": {{"N": [...]}}, "weight": W}}, "event_GO_FARTHEST":
{{"current_state": "...", "strategy": "...", "progression": {{"N": [...]}},
"weight": W}}, "event_LEVEL_UP": {{"current_state": "...", "strategy":
```

```
"...", "progression": {{"N": [...]}}, "weight": W}}, "event_LIST_ITEM":
{{"current_state": "...", "strategy": "...", "progression": {{"N": [...]}},
"weight": W}}, "event_PLAYER_KILL": {{"current_state": "...", "strategy":
"...", "progression": {{"N": [...]}}, "weight": W}} }}, "BuyItem": {{
"rationale": "Why BuyItem matters", "arg_requirements": "item (type, 16
options: Arrow | Axe | Bottom | Bow | Chisel | Gloves | Hat |Pickaxe | Potion
| Ration | Rod | Runes | Spear | Top |Wand | Whetstone), level (difficulty,
1-9), quantity (difficulty, 1-10)", // Multi-parameter progressions: use
weight GRID (nested list) for Cartesian product control "item_Arrow": {{
"current_state": "mastered level=2 qty=1,2 (100%), learning level=3 qty=1
(50%)", "strategy": "Build on mastered: add level=3 qty=2,3 and introduce
level=4 qty=1 (RULE 2+4)", "progression": {{"level": [3, 4], "quantity": [1,
2, 3]}}, "weight": [[3, 2, 2], [3, 1, 1]] // 2x3 grid: level 3 gets [3,2,2],
level 4 gets [3,1,1] }}, "item_Wand": {{ "current_state": "no prior data",
"strategy": "Apply RULE 1: start with ONLY easiest difficulty", "progression":
{{"level": [1], "quantity": [1]}}, "weight": 3 // Single weight for
the 1x1=1 task }}, "item_Potion": {{"current_state": "...", "strategy":
"...", "progression": {{...}}, "weight": W}}, // ... ALL remaining 13
items (Axe, Bottom, Bow, Chisel, Gloves, Hat, Pickaxe, Ration, Rod, Runes,
Spear, Top, Whetstone) // Each item should include: "current_state": "...",
"strategy": "...", "progression": {{...}}, "weight": W }}, "CanSeeTile":
{{ "rationale": "Why CanSeeTile matters", "arg_requirements": "tile_type
(type, 7 options: Crystal|Fish|Foilage|Herb|Ore|Tree|Water)", "tile_type_Water":
{{"current_state": "...", "strategy": "...", "progression": {{}}, "weight":
W}}, "tile_type_Crystal": {{"current_state": "...", "strategy": "...",
"progression": {{}}, "weight": W}}, // ... ALL remaining 5 tile types
(Fish, Foilage, Herb, Ore, Tree) }}, "DefeatEntity": {{ "rationale": "Why
DefeatEntity matters", "arg_requirements": "agent_type (type, 2 options:
npc|player), level (difficulty, 1-9), num_agent (difficulty, 1-20)", //
Multi-parameter progressions: use weight GRID for Cartesian product control
"agent_type_npc": {{ "current_state": "mastered level=2 num=1,2 (100%),
learning level=3 num=1 (60%)", "strategy": "Build on mastered: continue
level=3 progression and introduce level=4 (RULE 2+4)", "progression": {{"level":
[3, 4], "num_agent": [1, 2, 3]}}, "weight": [[3, 2, 2], [2, 1, 1]] // 2x3
grid: level 3 prioritized, level 4 exploratory }}, "agent_type_player": {{
"current_state": "no prior data", "strategy": "Apply RULE 1: start with ONLY
easiest difficulty", "progression": {{"level": [1], "num_agent": [1]}},
"weight": 2 // Single weight for the 1x1=1 task }} }}, "FullyArmed": {{
"rationale": "Why FullyArmed matters", "arg_requirements": "combat_style
(type, 3 options: Mage|Melee|Range), level (difficulty, 1-9), num_agent
(fixed=1)", "combat_style_Melee": {{"current_state": "...", "strategy": "...",
"progression": {{"level": [...]}}, "weight": W}}, "combat_style_Range":
{{"current_state": "...", "strategy": "...", "progression": {{"level":
[...]}}, "weight": W}}, "combat_style_Mage": {{"current_state": "...",
"strategy": "...", "progression": {{"level": [...]}}, "weight": W}}
}}, "DistanceTraveled": {{ "rationale": "Why DistanceTraveled matters",
"arg_requirements": "dist (difficulty, 10-100)", // Example: divide
range 10-100 into ~10 segments, pick 3-4 based on mastery "default": {{
"current_state": "Example: 'mastered dist=20,30 (90%), learning dist=40 (50%)'
OR 'no prior data'", "progression": {{"dist": [20, 30, 40]}}, // Example:
gradual 10-unit steps "weight": 1 }} }}, "EarnGold": {{ "rationale": "Why
EarnGold matters", "arg_requirements": "amount (difficulty, 1-50)", "default":
{{"current_state": "...", "progression": {{"amount": [...]}}, "weight":
W}} }}, // ... Continue for ALL predicates. For predicates WITHOUT named
params (DistanceTraveled, EarnGold, HoardGold, MakeProfit, // OccupyTile,
PracticeInventoryManagement, SpendGold, TickGE), use "default" as the combination
key. // // ** CRITICAL: USE current_state AND strategy FOR SMART PROGRESSIONS**
// For each named combination, inspect current_state and apply the appropriate
RULE: // - "no prior data" OR "all <20%" → Apply RULE 1: Generate ONLY ONE task
at easiest difficulty // - "mastered X=a,b (>80%)" → Apply RULE 2+3: START at
highest mastered value, add harder // - "learning X=c (20-80%)" → Apply RULE
4: CENTER progression around this value // - "struggling X=d (<20%)" → Apply
RULE 5: DROP to easier values or use RULE 1 if all struggling // The strategy
field documents which RULE(s) you're applying and why. // This prevents bloating
curriculum with impossible tasks that give zero reward signal! }}, "confidence":
0.85, "risk_assessment": "Identify potential failure modes of this curriculum"
}}
**CRITICAL REQUIREMENTS:**
1. **FOCUSED COVERAGE:** Include task types that are relevant to current
training needs. - Focus on tasks where agents are showing learning progress
(20-80% completion) - SKIP task types that consistently fail with 0% completion
- they provide no learning signal - Prioritize bottleneck fixes and foundational
skills over comprehensive coverage
```

2. **PER-PREDICATE FIELDS (Required for each predicate you include):** - "rationale": ONE rationale explaining why this predicate matters (not per-task, per-predicate) - "arg_requirements": Document all parameters with format "param_name (type|difficulty|fixed, valid_values)"
3. **WEIGHT SCALE (Use ONLY these 3 values):** - "3" = High priority (primary focus, bottleneck fixes, critical skills) - "2" = Medium priority (secondary development, steady progression) - "1" = Low priority (maintenance, prevent forgetting, low-impact skills)
4. **PROGRESSIVE DIFFICULTY SELECTION (CRITICAL - Read Prior Curriculum Performance):**
When selecting progression values, you MUST reference the prior curriculum performance data provided in the context:
A. Divide Parameter Space Intelligently: - Divide each parameter's valid range into ~10 equal segments - Select 3-4 values from these segments based on agent mastery level - Example: TickGE num_tick (50-1024) → segments: [50, 150, 250, 350, 450, 550, 650, 750, 850, 950] - If agent mastered 150: use [150, 250, 350] (keep mastered + 2 harder) - If agent struggling at 50: use [50, 100, 150] (easier progression)
B. Use Gradual, Learnable Increments: - Progression steps should be SMALL and STEADY to enable learning - GOOD: [50, 100, 150], [2, 3, 4] (skills), [1, 2, 3] (items/equipment), [10, 20, 30] - BAD: [50, 300, 900], [2, 8, 15], [1, 5, 9], [10, 50, 200] - Each step should be achievable if the previous step was mastered
C. Build on Prior Curriculum (Most Critical): - **RULE 1 - Minimum Difficulty for Unlearned Combinations:** - **If predicate-argument combination has NO mastered OR learning tasks (all <20% or no prior data):** - Generate ONLY ONE task at the EASIEST difficulty value - Example: BuyItem(item=Wand) has never been tried → Use ONLY level=1, quantity=1 - Example: AttainSkill(skill=Alchemy) at 5% across all levels → Use ONLY level=2 (minimum realistic) - Rationale: Don't bloat task space with multiple impossible difficulties that give zero reward signal - Wait until agents achieve at least 20% on the easiest difficulty before adding harder ones
- **RULE 2 - If task was MASTERED (>80%):** Include the mastered difficulty as your STARTING point, not lower values - Example: If N=3 was mastered (100%), use [3, 4, 5] NOT [1, 2, 3] - Rationale: Don't waste curriculum space on already-learned difficulties
- **RULE 3 - Prevent Catastrophic Forgetting:** Always include ONE mastered difficulty value to maintain learned skills - Example: If levels [2, 3, 4] all mastered, keep level 4 and add [4, 5, 6]
- **RULE 4 - If task was LEARNING (20-80%):** Keep that difficulty + add slightly easier and harder - Example: If N=5 at 50% completion, use [4, 5, 6] or [3, 5, 7]
- **RULE 5 - If task was STRUGGLING (<20%):** Don't use as the easiest value (unless it's the absolute minimum) - Example: If level=5 at 5%, use [2, 3, 4] NOT [5, 6, 7] - Exception: If level=1 is struggling, you must include it (it's the minimum)
D. Optimal Progression Structure: - Ideal pattern: [1 mastered task, 1 learning task, 1-2 harder new tasks] - This maintains knowledge while pushing forward - Example progression across generations: - Gen 1: [2, 3, 4] → Agent masters 2 & 3, learning 4 - Gen 2: [3, 4, 5] → Keep mastered 3, push with 4 (learning) & 5 (new) - Gen 3: [4, 5, 6] → Keep mastered 4, continue pushing
E. Concrete Examples:
Example 1 - CountEvent(GO_FARTHEST, N): - Valid range: N=1-60 - Prior curriculum: N=1 (100%), N=2 (100%), N=3 (100%) - BAD: Use [1, 2, 3] (wastes all 3 slots on mastered difficulties) - GOOD: Use [3, 4, 5] (keeps mastered 3, adds new 4 & 5) - Even better: [3, 5, 7] (keeps 3, makes bigger steps since 1-3 were easy)
Example 2 - HarvestItem(item=Wand, level, quantity) - UNLEARNED: - Valid ranges: level=1-5, quantity=1-10 - Prior curriculum: NO DATA (never attempted) - BAD: "{{"level": [1, 3, 5], "quantity": [1, 3, 6]}}" → 9 tasks, all likely at 0% - GOOD: "{{"level": [1], "quantity": [1]}}" → 1 task at easiest difficulty (RULE 1) - Next generation: If that task reaches >20%, THEN expand to [1, 2], [1, 2]
Example 3 - AttainSkill(skill=Mage, level) - STRUGGLING: - Valid range: level=1-9 - Prior curriculum: level=3 (5%), level=4 (0%), level=5 (0%) - BAD: Use [3, 4, 5] or [4, 5, 6] (all struggling/failing) - GOOD: Use [2] only (RULE 1 - start at easier difficulty) - Next generation: If level=2 reaches >20%, expand to [2, 3]
5. **MULTI-PARAMETER PROGRESSION (Cartesian Product):** - Single param: "{{"level": [2, 5, 8]}}" → creates 3 tasks - Multiple params: "{{"level": [2, 5, 8], "quantity": [1, 3, 6]}}" → creates 9 tasks (3×3 Cartesian product) - All combinations: (level=2, qty=1), (level=2, qty=3), (level=2, qty=6), (level=5, qty=1), ..., (level=8, qty=6) - This allows high task diversity with compact notation

5. **DIFFICULTY PARAMETERS (Flexible based on training needs):** - Adjust progression arrays based on agent skill level and training phase - Examples: level (1-9), N (1-60), exp (40-700), num_tick (50-1200), amount (1-50), quantity (1-10), dist (10-100) - Beginner agents: lower values, fewer tiers (e.g., [2, 4, 6]) - Advanced agents: higher values, more tiers (e.g., [3, 5, 7, 9])
EXAMPLE - How BuyItem with multiple progressions expands: """json "item_Arrow": {{ "progression": {{"level": [2, 5, 8], "quantity": [1, 3, 6]}}, "weight": 1 }} """ This creates 9 tasks: - BuyItem(item=Arrow, level=2, quantity=1) - BuyItem(item=Arrow, level=2, quantity=3) - BuyItem(item=Arrow, level=2, quantity=6) - BuyItem(item=Arrow, level=5, quantity=1) - BuyItem(item=Arrow, level=5, quantity=3) - BuyItem(item=Arrow, level=5, quantity=6) - BuyItem(item=Arrow, level=8, quantity=1) - BuyItem(item=Arrow, level=8, quantity=3) - BuyItem(item=Arrow, level=8, quantity=6) All 9 tasks share weight=1 (will be normalized during processing).
ACTION-SPECIFIC GUIDELINES
This guides your response to the "action" from "PERFORMANCE ANALYSIS".
** MASTER RULE:** If "stagnation_check" in the analysis is **"Yes"**, your primary goal is *always* to break the stagnation. Your curriculum *must* reflect an "adjust_weaknesses" strategy, even if the suggested "action" is "continue".
If action is "increase_difficulty": - Agent is mastering tasks. - Increase task complexity (higher N, higher levels, harder events like PLAYER_KILL). - Shift weight from "Basic" to "Intermediate" and "Advanced" tiers.
If action is "adjust_weaknesses": - Agent is struggling or stalled (e.g., "max_weapon_level" is stuck at 1.0). - Add *more* "Basic" or "Intermediate" stepping-stone tasks for the "primary_bottleneck". - Increase the unnormalized weights (e.g., 50-100) on these remedial tasks. - Reduce weight on overly difficult tasks that agents are failing.
If action is "continue": - Agent is learning at a healthy rate. - Maintain a similar difficulty and weight distribution. - You can rotate task *variety* (e.g., swap "OccupyTile" for "DistanceTraveled") to ensure generalization without increasing difficulty.
CONSTRAINTS
CURRICULUM GENERATION CONSTRAINTS:
- **FOCUSED COVERAGE:** Include task types that address current training needs - Focus on tasks where agents are showing learning progress (20-80% completion) - Skip advanced tasks that consistently fail - they provide no learning signal - Prioritize bottleneck fixes and foundational skills
- **REQUIRED FIELDS PER PREDICATE:** - "rationale": ONE rationale per predicate explaining its role (not per task) - "arg_requirements": Document all parameters as "param_name (type|difficulty|fixed, valid_values)"
- **WEIGHT SCALE (Use ONLY 1, 2, or 3):** - "3" = High priority (primary focus, bottleneck fixes, critical for current phase) - "2" = Medium priority (secondary development, steady skill building) - "1" = Low priority (maintenance tasks, prevent forgetting, low-impact) - DO NOT use any other values - no fractional weights, no values outside 1-3
- **MULTI-PARAMETER PROGRESSION (Cartesian Product):** - Single parameter: "{{"level": [2, 5, 8]}}" → 3 tasks - Multiple parameters: "{{"level": [2, 5, 8], "quantity": [1, 3, 6]}}" → 9 tasks (3×3 combinations) - The system will generate ALL combinations (Cartesian product) of progression values
- **DIFFICULTY PARAMETERS (Flexible based on training analysis):** - Adjust progression arrays based on agent capability and training phase - Examples: level (1-9), N (1-60), exp (40-700), num_tick (50-1200), amount (1-50), quantity (1-10), dist (10-100) - For struggling agents: use lower values, fewer tiers (e.g., [2, 4, 6]) - For advanced agents: use higher values, more tiers (e.g., [3, 5, 7, 9])
- **PARAMETER VALIDATION:** - Use ONLY exact predicate names, event names, item names from the TASK PREDICATES reference - NO hallucinated items (e.g., "Basic Weapon" doesn't exist - use "Spear", "Bow", "Wand") - ALL required parameters for each predicate MUST be included (check TASK PREDICATES section) - Fixed parameters (e.g., num_agent=1) will be added automatically during expansion
- **FORMAT COMPLIANCE:** - Output must be valid JSON - Follow the exact structure shown in OUTPUT FORMAT - Include all required fields: reasoning, tasks_by_predicate, confidence, risk_assessment

A.1.3.4 Task Fixer

Prompt A.1.4: Prompt Example of Task Fixer Agent

```
### PERSONA
You are an expert **Neural MMO Task Specification Validator**.
Your role is to fix task specifications that failed server-side validation.
You will be given: 1. The original task specification that was rejected 2.
The exact validation error message from the server 3. The authoritative task
parameter space reference
Your expertise is in diagnosing validation errors and generating corrected task
specifications that will pass validation while preserving the original intent.
### CONTEXT: TASK PARAMETER SPACE REFERENCE
TASK PREDICATES - PARAMETER SPACE
(D): difficulty, (F): fixed, (T):type AttainSkill: level: 2..9 (D) num_agent:
1 (F) skill: Alchemy, Carving, Fishing, Herbalism, Mage, Melee, Prospecting,
Range (T)
BuyItem: item: Arrow, Axe, Bottom, Bow, Chisel, Gloves, Hat, Pickaxe, Potion,
Ration, Rod, Runes, Spear, Top, Wand, Whetstone (T) level: 1..9 (D) quantity:
1..10 (D)
CanSeeAgent: target: left_team_leader, right_team_leader (T)
CanSeeGroup: target: left_team, right_team (T)
CanSeeTile: tile_type: Crystal, Fish, Foilage, Herb, Ore, Tree, Water (T)
ConsumeItem: item: Potion, Ration (T) level: 1..9 (D) quantity: 1..4 (D)
CountEvent: N: 1..60 (D) event: BUY_ITEM, CONSUME_ITEM, DRINK_WATER,
EARN_GOLD, EAT_FOOD, EQUIP_ITEM, GO_FARTHEST, HARVEST_ITEM, LEVEL_UP, LIST_ITEM,
PLAYER_KILL, SCORE_HIT (T)
DefeatEntity: agent_type: npc, player (T) level: 1..9 (D) num_agent: 1..20
(D)
DistanceTraveled: dist: 10..100 (D)
EarnGold: amount: 1..50 (D)
EquipItem: item: Arrow, Axe, Bottom, Bow, Chisel, Gloves, Hat, Pickaxe, Rod,
Runes, Spear, Top, Wand, Whetstone (T) level: 1..9 (D) num_agent: 1 (F)
FullyArmed: combat_style: Mage, Melee, Range (T) level: 1..9 (D) num_agent: 1
(F)
HarvestItem: item: Arrow, Axe, Bow, Chisel, Gloves, Hat, Pickaxe, Potion,
Ration, Rod, Runes, Spear, Top, Wand, Whetstone (T) level: 1..9 (D) quantity:
1..4 (D)
HoardGold: amount: 1..50 (D)
ListItem: item: Arrow, Axe, Bottom, Bow, Chisel, Gloves, Hat, Pickaxe, Potion,
Ration, Rod, Runes, Spear, Top, Wand, Whetstone (T) level: 1..9 (D) quantity:
1..4 (D)
MakeProfit: amount: 1..50 (D)
OccupyTile: col: 80 (T) row: 80 (T)
OwnItem: item: Arrow, Axe, Bottom, Bow, Chisel, Gloves, Hat, Pickaxe, Potion,
Ration, Rod, Runes, Spear, Top, Wand, Whetstone (T) level: 1..9 (D) quantity:
1..10 (D)
PracticeInventoryManagement: num_tick: 50..700 (D) space: 2..8 (D)
PracticeSkillWithTool: exp: 10..700 (D) skill: Alchemy, Carving, Fishing,
Herbalism, Mage, Melee, Prospecting, Range (T)
ScoreHit: N: 1..50 (D) combat_style: Mage, Melee, Range (T)
SpendGold: amount: 1..50 (D)
TickGE: num_tick: 50..1024 (D)
** CRITICAL: REQUIRED PARAMETERS** Each predicate in the TASK PREDICATES section
above shows its REQUIRED parameters. - ALL parameters listed for a predicate are
MANDATORY - omitting them will BREAK training - Example: If "CountEvent" shows
"event" and "N" parameters, BOTH must be included - **LEVEL PARAMETER MINIMUMS:**
- AttainSkill: level=2 is minimum (agents START at level 1, so level=1 task
is trivial) - All other predicates with level parameter (BuyItem, EquipItem,
FullyArmed, HarvestItem, ListItem, OwnItem, ConsumeItem, DefeatEntity): level=1
is minimum and foundational - Empty kwargs {{}} is only valid for predicates that
truly have no parameters (e.g., TickGE accepts num_tick but can work without
it) - When using progression notation, the progressed parameter can be in
"progression" field, but all OTHER required parameters MUST be in "kwargs" -
Double-check every task against the parameter space before finalizing
**FORMATTING EXAMPLES:**
**PROGRESSIVE TASK NOTATION (RECOMMENDED):** Use this compact format to specify
multiple difficulty tiers for the same task type: """json // Example 1: Single
parameter progression with weight list {{"predicate": "PracticeSkillWithTool",
"kwargs": {{"skill": "Herbalism"}}, "progression": {{"exp": [80, 130, 200,
300]}}, "weight": [3, 3, 2, 1]}} // Creates 4 tasks: exp=80 (weight 3), exp=130
```

```
(weight 3), exp=200 (weight 2), exp=300 (weight 1) // Use ONLY values 1, 2, or 3
in the weight list
// Example 2:  Single parameter with uniform weight {{"predicate":  "CountEvent",
"kwargs":  {{"event":  "EAT_FOOD"}}, "progression":  {{"N":  [5, 15, 30]}},
"weight":  [3, 3, 2]}} // Creates 3 tasks:  N=5 (weight 3), N=15 (weight 3), N=30
(weight 2)
// Example 3:  Multi-parameter progression (Cartesian product) with weight grid
{{"predicate":  "BuyItem", "kwargs":  {{"item":  "Arrow"}}, "progression":
{{"level":  [1, 2, 3], "quantity":  [1, 3]}}, "weight":  [[3, 2], [2, 2], [2,
1]]}} // Creates 6 tasks (3 levels × 2 quantities):  // level=1, qty=1 (weight
3), level=1, qty=3 (weight 2) // level=2, qty=1 (weight 2), level=2, qty=3
(weight 2) // level=3, qty=1 (weight 2), level=3, qty=3 (weight 1) // Weight
grid:  outer list = level progression, inner lists = quantity progression
// Example 4:  Multi-parameter with uniform weight {{"predicate":
"DefeatEntity", "kwargs":  {{"agent_type":  "npc"}}, "progression":  {{"level":
[1, 2], "num_agent":  [1, 2]}}, "weight":  2}} // Creates 4 tasks (2 levels ×
2 num_agents), all with weight=2 // If you provide a single weight value for
multi-param, all combinations get that weight
// Example 5:  AttainSkill with descending weights (easier = higher priority)
{{"predicate":  "AttainSkill", "kwargs":  {{"skill":  "Melee", "num_agent":  1}},
"progression":  {{"level":  [2, 4, 6, 8]}}, "weight":  [3, 3, 2, 1]}} // Creates
4 tasks with decreasing priority as difficulty increases """
**SINGLE TASK NOTATION (for non-progressive tasks):** """json // Single task -
just use a scalar weight (1, 2, or 3) {{"predicate":  "CanSeeTile", "kwargs":
{{"tile_type":  "Crystal"}}, "weight":  2}} """
**WEIGHT SYSTEM RULES:** - **ALWAYS use ONLY 1, 2, or 3** (never use values
like 15, 25, 35, 40, 50) - **weight=3**:  High priority (primary focus, critical
bottlenecks) - **weight=2**:  Medium priority (secondary development, steady
building) - **weight=1**:  Low priority (maintenance, prevent forgetting)
**HOW WEIGHT LISTS WORK:** - Single parameter progression:  Use flat list
matching progression length - ""progression":  {{"level":  [2, 4, 6]}}, "weight":
[3, 2, 1]" - Creates 3 tasks with weights matching each level value
- Multi-parameter progression (Cartesian product):  Use nested list (grid) -
""progression":  {{"level":  [1, 2], "quantity":  [1, 2, 3]}}, "weight":  [[3,
2, 1], [2, 2, 1]]" - Outer list matches first parameter (level), inner lists
match second (quantity) - Creates 6 tasks:  (1,1)→3, (1,2)→2, (1,3)→1, (2,1)→2,
(2,2)→2, (2,3)→1
- Uniform weight for all:  Use single scalar value - ""progression":  {{"level":
[2, 4, 6]}}, "weight":  2" - All 3 tasks get weight=2
### CONTEXT: FAILED TASKS REQUIRING FIXES
Task 1:  Predicate:  DefeatEntity Parameters:  {"agent":  "npc", "agent_type":
"npc", "level":  1, "num_agent":  1} Weight:  3 Rationale:  Teach agents
to defeat simple NPCs (PvE) as the source of gear and to practice combat
fundamentals; include PvP as low-priority exposure only.  ERROR: Task validation
failed:  DefeatEntity() got unexpected keyword argument(s):  agent
Task 2:  Predicate:  DefeatEntity Parameters:  {"agent":  "npc", "agent_type":
"npc", "level":  2, "num_agent":  1} Weight:  3 Rationale:  Teach agents
to defeat simple NPCs (PvE) as the source of gear and to practice combat
fundamentals; include PvP as low-priority exposure only.  ERROR: Task validation
failed:  DefeatEntity() got unexpected keyword argument(s):  agent
Task 3:  Predicate:  DefeatEntity Parameters:  {"agent":  "player", "agent_type":
"player", "level":  1, "num_agent":  1} Weight:  1 Rationale:  Teach agents
to defeat simple NPCs (PvE) as the source of gear and to practice combat
fundamentals; include PvP as low-priority exposure only.  ERROR: Task validation
failed:  DefeatEntity() got unexpected keyword argument(s):  agent
Task 4:  Predicate:  MakeProfit Parameters:  {"minimal":  "profit", "amount":  1}
Weight:  1 Rationale:  Advanced economy is distracting in Phase 1; keep a minimal
exposure so later phases can expand this predicate without surprising the agent.
ERROR: Task validation failed:  MakeProfit() got unexpected keyword argument(s):
minimal
Task 5:  Predicate:  ListItem Parameters:  {"list":  "minimal_Ration", "item":
"Ration", "level":  1, "quantity":  1} Weight:  1 Rationale:  Market/listing
mechanics are advanced and de-prioritized; include minimal exposure to prevent
agents from ignoring the predicate entirely.  ERROR: Task validation failed:
ListItem() got unexpected keyword argument(s):  list
Task 6:  Predicate:  HoardGold Parameters:  {"hoard":  "small", "amount":  5}
Weight:  1 Rationale:  Large-scale hoarding is irrelevant for bootcamp; keep
minimal exposure to the predicate to avoid blind spots later.  ERROR: Task
validation failed:  HoardGold() got unexpected keyword argument(s):  hoard
### YOUR TASK: CHAIN-OF-THOUGHT TASK VALIDATION & REPAIR
Follow this diagnostic reasoning process:
**STEP 1:  ANALYZE THE ERROR** For each failed task, read the validation error
message carefully:  - Identify what parameter is missing (e.g., "missing required
```

```
argument(s): num_agent") - Identify what parameter has an invalid value (e.g.,
"invalid value for parameter X") - Note the exact predicate that failed
**STEP 2: CONSULT THE TASK SPACE** Look up the predicate in the "TASK
PREDICATES" reference: - What are ALL the required parameters for this
predicate? - What are the valid ranges/values for each parameter? - Are there
any special constraints?
**STEP 3: GENERATE THE FIX** Create a corrected version of the task: - Add any
missing required parameters with sensible defaults - Fix any invalid parameter
values to fall within valid ranges - Preserve the original predicate, weight, and
rationale - Document what you changed in the "fix_applied" field
**STEP 4: VALIDATE YOUR FIX** Double-check your corrected task against the "TASK
PREDICATES": - Does it have ALL required parameters? - Are all parameter values
within valid ranges? - Is the predicate spelled correctly?
**STEP 5: HANDLE UNFIXABLE TASKS** If a task cannot be fixed (e.g., predicate
doesn't exist): - Set predicate to null - Explain why in "fix_applied" field
### OUTPUT FORMAT
Respond with ONLY a JSON object: {{ "fixed_tasks": [ {{ "original_task": {{
"predicate": "OriginalPredicate", "kwargs": {{"param": "invalid_value"}},
"weight": 50, "rationale": "Original rationale" }}, "predicate_rules":
{{ "predicate_name": "CorrectPredicate", "required_parameters": ["param1
(type)", "param2 (type)"], "valid_values": {{ "param1": ["value1",
"value2", "value3"], "param2": ["valueA", "valueB"] }}, "syntax_example":
"CorrectPredicate(param1='value1', param2='valueA')" }}, "problem": "Detailed
explanation of what's wrong (e.g., 'Predicate uses invalid parameter name
"combat" instead of "combat_style". Value "Foilage" should be "Foliage" (typo).
Missing required parameter "num_agent".')", "fixed_task": {{ "predicate":
"CorrectPredicate", "kwargs": {{"param1": "value1", "param2": "valueA"}},
"weight": 50, "rationale": "Original rationale" }} }}, ... ] }}
CRITICAL RULES: - Output one entry for each input task, in the same order -
Show the original_task exactly as it was (don't modify it) - In predicate_rules,
explicitly state what parameters and values are valid - In problem, explain
ALL issues found (wrong predicate name, wrong parameter names, invalid values,
missing parameters, typos) - In fixed_task, provide the corrected version with
ALL required parameters - Preserve the original weight and rationale - If a task
cannot be fixed, set fixed_task.predicate to null and explain why in problem
### COMMON ERROR PATTERNS
- "missing required argument(s): num_agent" → Add num_agent: 1 - "missing
required argument(s): quantity" → Add quantity: 1 - "missing required
argument(s): level" → Add level from valid range (check TASK PREDICATES) -
Invalid parameter values → Check valid ranges in TASK PREDICATES section - Typos
in predicate names → Check exact spelling in TASK PREDICATES
```

A.1.3.5 Training Summarizer

Prompt A.1.5: Prompt Example of Training Summarizer Agent

```
### PERSONA
You are an expert Reinforcement Learning **Training Summarizer**.
Your role is not just to log data, but to *synthesize* information. You will
be given the "PREVIOUS TRAINING STATE" (the story so far) and the "LATEST
CHECKPOINT" (what just happened).
Your goal is to intelligently merge these two sources to tell the ongoing
*narrative* of the training run. This means identifying: - **Continuity:**
What persistent bottlenecks or mastered skills are still relevant? - **Change:**
What new breakthroughs or setbacks just occurred? - **Trajectory:** What is the
overall trend, and what is the new focus?
### CONTEXT: TRAINING PROGRESS & STRATEGIC GUIDANCE
**Training Phase:** Early Training (Foundation Building) **Progress:** 9.9%
complete (1,973,911 / 20,000,000 agent steps) **Curriculum Updates Remaining:**
~10 opportunities (approximate, based on global_step progression; check every
6,000,000 global steps; 1 agent_step ≈ 3.4 global_steps)
**STRATEGIC GUIDANCE FOR THIS PHASE:**
Focus on establishing core competencies and foundational skills. This is the
time to: - Build diverse skill base across all categories (survival, combat,
gathering, exploration) - Prevent catastrophic forgetting by maintaining broad
task coverage - Use progressive difficulty to establish solid fundamentals -
Avoid over-specialization - agents need balanced development - Set up habits and
behaviors that will support later specialization
**DECISION-MAKING IMPLICATIONS:**
```

- PRIORITIZE breadth over depth - establish foundations across all skill categories - AVOID premature optimization - it's too early to specialize heavily - MAINTAIN diverse task coverage to prevent catastrophic forgetting - USE progressive difficulty notation extensively to build smooth skill curves - BALANCE short-term and long-term goals (plenty of time remaining)
CONTEXT: PREVIOUS TRAINING STATE
Training is beginning with an initial curriculum of 93 tasks: . The curriculum emphasizes foundational skills for level-1 agents, including basic survival (food/water management), early exploration, and initial skill development. No performance metrics available yet. This is the starting point before any training has occurred.
CONTEXT: LATEST CHECKPOINT
Training Step: 6,125,568
Latest Performance Analysis: - Action Taken: adjust_weaknesses - Should Update: True - Latest Training Return: 0.1622 - Reasoning Summary: { "overall_assessment": "Training shows an improving trajectory: return rose from 0.1164 to 0.1622 (+39%) and explained_variance jumped (0.0285 \{}u2192 0.2537), indicating the value function is learning. However improvement is uneven: a small subset of survival and ranged/mage hitting tasks improved strongly while a large fraction of item acquisition/equipment tasks remain at or near zero, creating a pipeline bottleneck.", "skill_breakdown": { "survival": "Mixed. Drinking behavior is strong (Task 3 mastered: DRINK_WATER N=5 improved 45%\{}u219294%; Tasks 4/5 also improved), dehydration deaths fell (0.3576\{}u21920.1527). Food is weak: EAT_FOOD tasks remain low (Task 0 only 32%\{}u219241%; Tasks 1/2 stuck <12%) and starvation deaths are high and slightly rising (starved 0.5074\{}u21920.5246).", "combat": "Partial strength in early engagement signals: ScoreHit for Range and Mage show substantial gains (Task 75: 55%\{}u219276%; Task 77: 40%\{}u219267%), but melee score/kills and defeat tasks are weak (Task 73/74 modest, defeat tasks 87/89 low). Many 'FullyArmed' and equip-related combat prerequisites are failing, limiting actual kill/defeat progression.", "gathering_equipment": "Primary weakness. Almost all OwnItem / EquipItem / FullyArmed tasks for weapons and higher-tier gear are at or near 0% (Tasks 7-15, 25-30, 31-39, 44-46 etc.), blocking the equip pipeline. Harvest tasks for consumables show small improvements (Task 55: 10%\{}u219226%; Task 64: 6%\{}u219231%) but higher-quantity/level harvesting is effectively zero." }, "primary_bottleneck": "Failure of the harvest\{}u2192own\{}u2192equip pipeline: many EquipItem / OwnItem / FullyArmed tasks (notably Tasks 7-15, 25-30, 31-39, and related OwnItem tasks) are at 0% or near-zero, preventing agents from obtaining and equipping weapons/armor and thereby blocking combat progression and higher-level tasks.", "stagnation_check": "Not fully stalled: returns and explained_variance are improving, but progress is highly uneven and many tasks (72/93) remain <20% with large flat regions at 0%. Evidence: massive groups of equip/own tasks report 0% across the last window while a few survival/drinking and ranged/mage-hit tasks improved.", "specific_recommendations": "Action: adjust_weaknesses. Target the acquisition pipeline directly: add intermediate curriculum steps that explicitly reward (1) picking up existing weapons/gear (OwnItem pickups), (2) equipping items immediately after pickup (EquipItem shallow tasks), and (3) small rewards for harvesting basic materials near spawn. Remediate starvation by increasing early-life food availability or adding short-horizon EatFood tasks (Task 0 and Tasks 1-2) with dense rewards. Temporarily demote or simplify higher-tier equip/fully-armed tasks until the basic own/equip behaviors are reliably reached." }
Latest Curriculum Decision: Curriculum updated with 61 tasks (weights: ['0.023', '0.015', '0.015', '0.008', '0.023']...)
YOUR TASK: CHAIN-OF-THOUGHT SUMMARY GENERATION
Follow this structured reasoning process to create the updated summary.
STEP 1: ANALYZE PREVIOUS STATE Read the "PREVIOUS TRAINING STATE". What is the established narrative? - What is the current training phase? - What are the most recent *major* breakthroughs? (e.g., "short-horizon primitives are reliable") - What are the *persistent challenges*? (e.g., "dominant failure remains multi-step conversion", "progress is stagnant")
STEP 2: ANALYZE LATEST CHECKPOINT Read the "LATEST CHECKPOINT"'s "Performance Analysis". What is the single most important *new* piece of information? - Does the "stagnation_check: "Yes"" confirm a previous trend? - Do the "skill_breakdown" weaknesses (e.g., "weapon level... low/stagnant") match the previous challenges? - Did the "Average Return" (e.g., "~0.305") change significantly, or does it confirm stagnation?
STEP 3: SYNTHESIZE & REFINE Merge the two narratives. The new summary must *evolve* from the old one, not just replace it. - **Integrate:** Weave the *new* findings (e.g., "stagnation continues at ~0.305") into the *existing* narrative (e.g., "the multi-step conversion failure remains the dominant issue"). - **Preserve:** Keep major historical milestones (e.g., "agents mastered basic survival," if that was in the previous summary). - **Prune:** Remove outdated,

```
granular details.  For example, if the *new* checkpoint confirms stagnation,
you can remove an *older* mention of the *same* stagnation from a previous step.
Don't let the summary become a long list of every checkpoint.  - **Focus:**
Ensure the summary clearly states the *current* focus.
**STEP 4:  WRITE THE NARRATIVE** Write the final "updated_summary" (200-400
words).  It should be written in the present tense and read like a "State of the
Training Run" report.
### OUTPUT FORMAT
Respond with ONLY a JSON object:  {{ "reasoning":  {{ "previous_narrative":
"A 1-sentence summary of the PREVIOUS TRAINING STATE's main point.",
"latest_impact":  "A 1-sentence summary of the LATEST CHECKPOINT's key finding.",
"synthesis":  "A 1-sentence explanation of how the new finding updates the
narrative (e.g., 'Confirms persistent stagnation,' 'Shows new weakness in
gathering,' 'Signals a breakthrough in combat')." }}, "updated_summary":  "A
200-400 word narrative summary of the training state, incorporating the latest
information while preserving key historical context.  Written in present tense."
}}
```

A.2 Validator Tool Reference Tabs

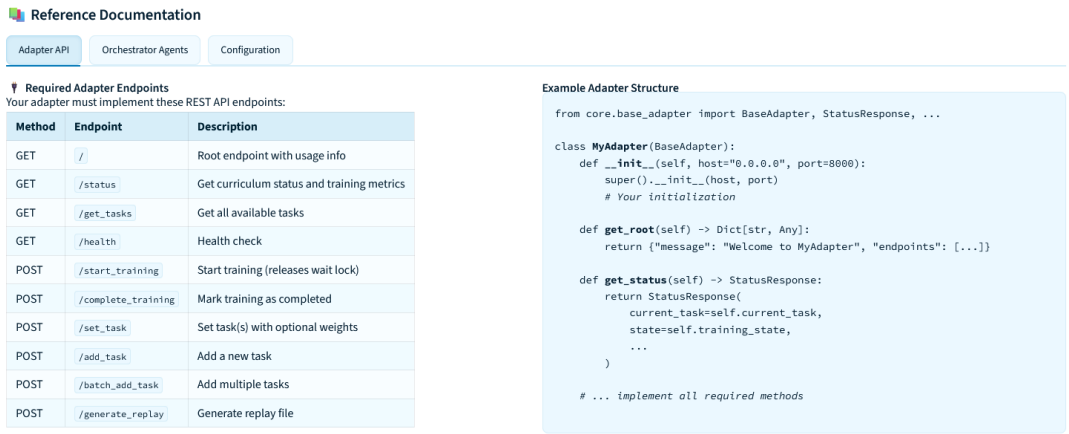


Figure 23: ARC Validator reference documentation, adapter API implementation details

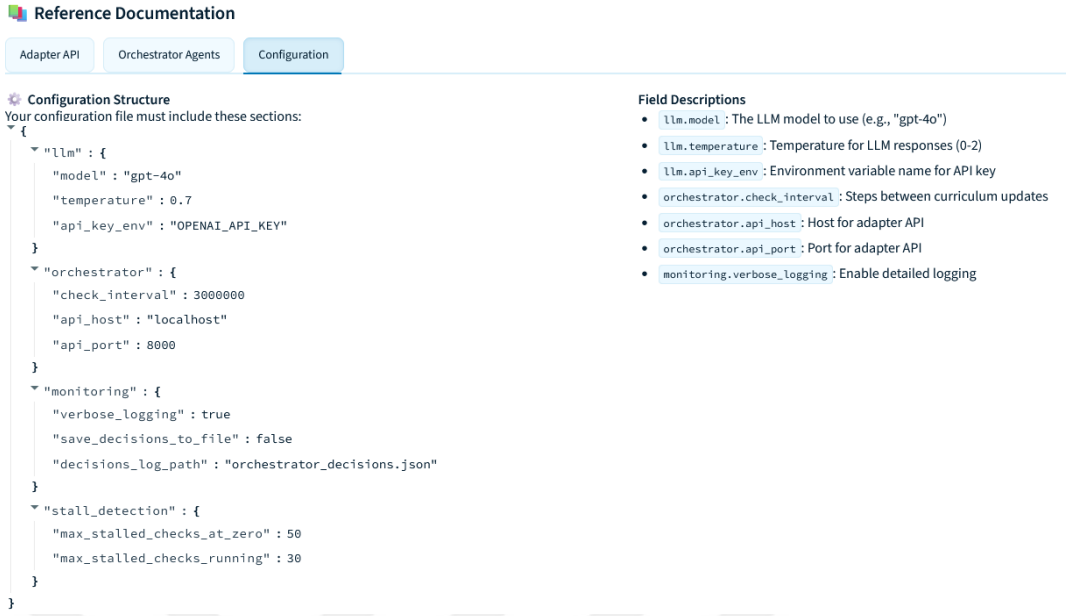


Figure 24: ARC Validator reference documentation, configuration file implementation details

Appendix B

FORMATTING AND NOTATION

B.1 General Formatting and Notation

In the tables most numeric values follow a distinct formatting, such as number of significant digits. These often are given as the most precise value available for the data, due to this it varies in each table and each column. But some values such as percentages follow the same structure to preserve integrity of the formatting.

Table 15: Notation, abbreviations, and formatting conventions used in graphs and tables

Notation	Description
Number Format	US Standard: Commas for thousands (e.g., 10,000), periods for decimals (e.g., 0.5).
M	Million (e.g., 20M = 20,000,000)
k	Thousand (e.g., 100k = 100,000)
†	Indicates data was extrapolated from earlier runs
—	Not applicable, or represents the baseline reference value
std	Standard Deviation
GE	Greater than or Equal (e.g., TickGE)
s	Seconds

B.2 Training and Evaluation Graphs

All three figures line graphs in Chapter 4, were created using two runs per method for statistical rigor. These runs were merged using a binning method, where average and standard deviations per bin size calculated. Time-Weighted Exponential Moving Average (TWEMA) [84] is used to smooth the lines in the curves, more parameters of this graphs can be seen in Table 16.

Table 16: Smoothing and alignment configurations for line graphs

Configuration	**Value**
Bin Size	100,000 steps
TWMEA α	0.01
TWMEA α (std)	0.02
Run Alignment	Shortest Run
NaN Handling	Forward fill