ENHANCING LARGE LANGUAGE MODELS WITH DECISION-RELEVANT KNOWLEDGE FOR REINFORCEMENT LEARNING TASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) have demonstrated strong language understanding and reasoning capabilities through large-scale pretraining, motivating their increasing use in reinforcement learning and sequential decision-making. However, pretrained LLMs lack sufficient understanding of task-specific decision-relevant knowledge, including critical state information and task rules that directly guide action selection. While such knowledge is essential for effective decision-making, general-purpose LLMs often fail to correctly interpret and leverage it in domain-specific reinforcement learning tasks. To address this gap, we propose a two-stage training framework that explicitly teaches LLMs task-specific decision-relevant knowledge before policy optimization. In Stage-A, we construct supervision signals from structured-state definitions and task rules to train the LLM to recognize key environmental facts and infer the current decision objective, yielding a decision representation h_t. In Stage-B, a lightweight policy is optimized with PPO on top of h_t through online interaction. Under the same online interaction budget, our method achieves better overall performance than the compared baselines on Atari and Crafter and also shows applicability to additional environments.

1 INTRODUCTION

Recent advances in large-scale text pretraining have endowed large language models (LLMs) with strong language understanding and reasoning capabilities, motivating their increasing use in reinforcement learning and sequential decision-making. Recent studies use LLMs to support exploration, reward construction, and learning from interaction experience in reinforcement learning (Hao et al., 2025; Qu et al., 2025; Zhang et al., 2026). Other work also studies LLM reasoning during policy optimization (Wang et al., 2026). At the same time, previous studies train pretrained LLMs on domain text or domain-specific data to improve their domain knowledge (Jiang et al., 2025; Choi et al., 2025; Yang et al., 2025b). Despite their strong general-purpose reasoning and language understanding capabilities, pretrained LLMs lack the domain knowledge required for specific decision-making tasks. Such knowledge is not limited to textual descriptions. It is frequently encoded in structured state information and task-specific rules. For example, in Atari games, observations indicating an enemy's position, distance, or number can be critical for assessing the current situation, while task rules specify how the agent should respond to such conditions, e.g., avoiding an enemy when it becomes dangerously close. Together, these state semantics and behavioral rules provide decision-relevant domain knowledge that can directly guide action selection. However, pretrained LLMs do not necessarily possess an accurate understanding of such task-specific knowledge. Simply providing structured state variables or task rules as additional inputs does not guarantee that the model can correctly interpret their semantics or relate them to appropriate decisions. In this paper, we study how to explicitly equip pretrained LLMs with decision-relevant domain knowledge before policy optimization. By enabling the LLM to interpret these forms of knowledge and encode them into a decision representation, our goal is to provide a stronger semantic foundation for downstream policy learning and improve decision-making performance in domain-specific reinforcement learning tasks.

We propose a two-stage training framework. In Stage-A, structured-state definitions and task rules are used to construct supervision signals that train the LLM to identify environmental facts and infer

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

the current decision objective. The LLM then encodes this decision-relevant domain knowledge into a decision representation h_t. In Stage-B, a lightweight policy is optimized with PPO on top of h_t through online environment interaction.

Overall, we make the following contributions:

(i) We formulate decision-relevant domain knowledge as an explicit learning objective before policy optimization. This knowledge includes the environmental facts encoded in structured states and the current decision objective inferred from these facts and the task rules.
(ii) We design two types of supervision that enable the LLM to learn decision-relevant domain knowledge and encode it into the decision representation h_t before policy optimization.
(iii) We propose a two-stage training framework that first learns decision-relevant domain knowledge and then uses the decision representation h_t for policy optimization.

2 Related Work

Large Language Models for Reinforcement Learning. Large language models have increasingly been used in reinforcement learning to provide task-related information during exploration, reward construction, and learning from interaction experience. LLM-Explorer (Hao et al., 2025) analyzes recent training trajectories and generates action distributions that guide subsequent exploration. LaRe (Qu et al., 2025) instead uses an LLM to generate executable latent-factor functions and constructs learned rewards from these factors. Agent Learning via Early Experience (Zhang et al., 2026) uses the agent's own early interaction experience to construct reflection data and then improves the policy through supervised fine-tuning and subsequent reinforcement learning. RAGEN-2 (Wang et al., 2026) studies how reasoning behavior evolves during multi-turn policy optimization. Text2Reward (Xie et al., 2024) and Eureka (Ma et al., 2024) further use LLMs to generate or refine reward functions for reinforcement learning.

Learning Domain-Specific Knowledge in Large Language Models. Adapting pretrained language models to domain-specific knowledge has been studied extensively. Gururangan et al. (2020) show that continued pretraining on in-domain text can improve downstream performance, establishing domain-adaptive pretraining as an effective approach for domain specialization. AdaptLLM (Cheng et al., 2024) further transforms domain corpora into reading-comprehension data to improve both domain knowledge acquisition and the use of this knowledge. More recent work explores different ways to improve domain knowledge learning. Mix-CPT (Jiang et al., 2025) separates domain knowledge learning from subsequent format alignment. Contextual Fine-Tuning (Choi et al., 2025) uses contextual information during training to improve domain knowledge learning. Synthetic Continued Pretraining (Yang et al., 2025b) constructs synthetic domain data for continued pretraining. Other work studies domain-specific fine-tuning and the identification and supplementation of missing knowledge in pretrained models (Lin et al., 2026; Xiong et al., 2026). These studies mainly acquire domain knowledge from textual corpora or domain-specific training data. Our work considers a different form of domain knowledge: decision-relevant knowledge expressed through structured states and task rules in reinforcement learning tasks.

Representation Learning for Decision Making. Representation learning has long been used to support downstream decision making. R3M (Nair et al., 2023) pretrains visual representations from large-scale human videos and uses the frozen representation for downstream policy learning. VIP (Ma et al., 2023) learns visual representations that can support downstream control and reward construction. More recent work studies how the internal representations of large models can be adapted to support downstream tasks. ReFT (Wu et al., 2024) learns task-specific interventions directly in the hidden representation space of language models, while PerFit (Liu et al., 2026) studies how personalization-related information is organized and adapted in the representation space of LLMs. Digi-Q (Bai et al., 2025) is more closely related to our setting. It first fine-tunes intermediate vision-language representations to make actionable information more accessible and then learns a Q-function from these representations. Our method is also based on learning a representation before downstream decision learning. Stage-A uses state-fact and rule-conditioned supervision to encode environmental facts and the current decision objective into h_t, and Stage-B then optimizes the policy using this representation.

3 METHOD

3.1 OVERVIEW

Our goal is to equip a pretrained LLM with decision-relevant domain knowledge before policy optimization and encode this knowledge into a decision representation for subsequent policy learning. To this end, we use a two-stage training framework. Given the current structured state, recent interaction history, and task rules, the model produces a decision representation h_t. Stage-A trains h_t to encode the environmental facts represented by the structured state and the current decision objective inferred from these facts and the task rules.

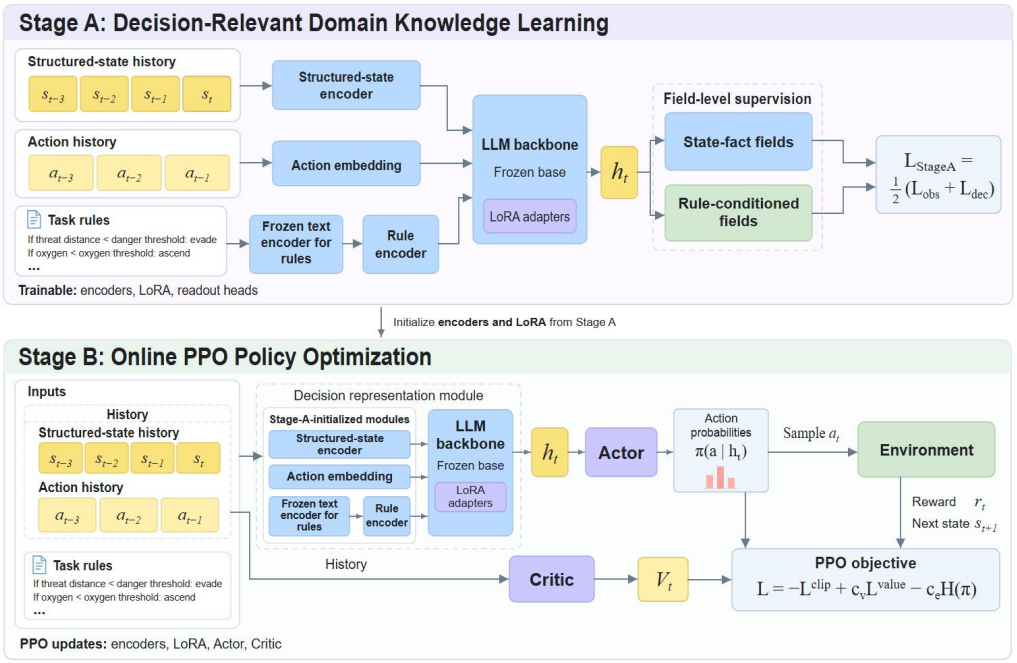


Figure 1: Overview of the proposed two-stage framework. Stage A (Top): State-fact and rule-conditioned field supervision train the LLM to encode decision-relevant domain knowledge from structured-state history, action history, and task rules. The learned encoders and LoRA adapters are then used to initialize Stage B. Stage B (Bottom): The actor uses the decision representation h_t for online policy learning, while the critic estimates the state value from interaction history. PPO updates the encoders, LoRA adapters, actor, and critic through environment interaction.

3.2 STAGE-A: DECISION-RELEVANT DOMAIN KNOWLEDGE LEARNING

State-Fact Fields. In domain-specific decision-making tasks such as Atari and Crafter, structured states often contain numerical variables that describe the current environment. These variables may represent object presence, positions, distances, or counts, and their meanings depend on the specific task. Stage-A therefore trains the LLM to interpret these variables and identify the environmental facts they represent. As shown in Figure 2, the structured state contains separate fields for up to four divers. Each field records whether the corresponding diver is present and its position. In this example, three of the four diver fields indicate visible divers, corresponding to the **environmental fact** that three divers are currently visible. We refer to this type of supervision as a state-fact field. Its

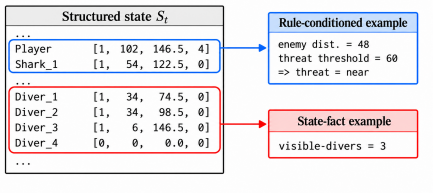


Figure 2: SeaQuest field examples.

label is derived directly from the structured state. When several numerical variables describe the same environmental fact, we use one state-fact field to supervise that fact in Stage-A.

Rule-Conditioned Fields. In addition to environmental facts that can be determined directly from the structured state, the interpretation of some state information depends on the current task rules. For example, as shown in Figure 3, in SeaQuest, the player is at $(102, 146.5)$, and a Shark is at $(54, 122.5)$. The threat-assessment rules identify this Shark as the current threat. Its horizontal distance from the player is 48. When the threat threshold is 60, the supervision-field label for the Shark–player distance is `near`. Therefore, the model needs both the structured state and the task rules to determine these labels. We call these supervision fields rule-conditioned fields. We determine their labels from both the structured state and the task rules. These discrete labels provide the rule-conditioned supervision targets for Stage-A. We also define a higher-level goal field to represent the **current decision objective**. In the example above, when the Shark–player distance label is `near`, the goal field is `survive`. The agent should then prioritize its own survival. When the distance label is `far`, the goal is to continue exploration. This supervision trains the LLM to interpret the structured state under the task rules and infer the current decision objective.



Figure 3: SeaQuest example showing how the threat distance obtained from the structured state and the task rule determine the current decision objective.

Field-Level Supervision. Field-level supervision trains the LLM to interpret the input information and encode it into h_t. This makes the relevant information easier to recover from h_t and supports subsequent policy learning in Stage-B. As shown in Figure 4, we attach an independent linear readout head (a single linear layer) to each supervision field using the decision representation h_t. Each readout head corresponds to one supervision field and predicts its discrete label. For example, the visible-divers head predicts the number of visible divers from h_t. The threat-distance head predicts whether the threat is `near` or `far` from h_t. These predictions are compared with the supervision labels generated from the structured state and task rules, and the resulting classification losses are used to update the trainable parameters involved in forming h_t. If a readout head predicts the wrong label, the corresponding supervision loss provides a correction signal that encourages h_t to better encode the relevant information.

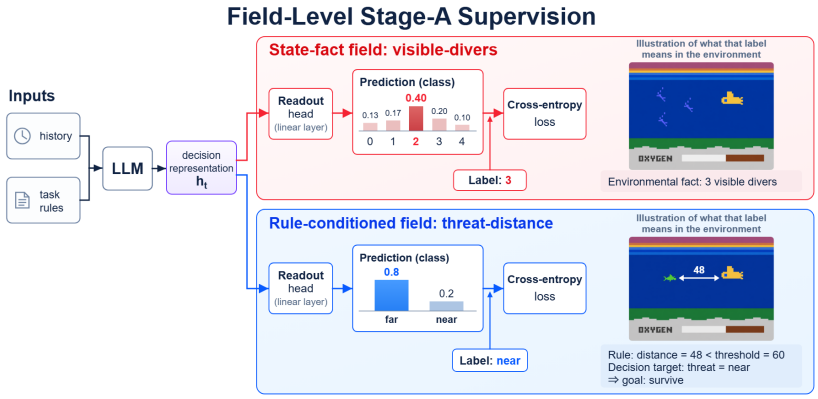


Figure 4: Example of field-level supervision in Stage-A. The LLM encodes the input history and task rules into the decision representation h_t. A separate linear readout head is attached to each state-fact or rule-conditioned field and predicts its discrete label from h_t. The prediction is compared with the corresponding supervision label using cross-entropy loss. As illustrated by the visible-divers and threat-distance examples, this supervision trains h_t to encode environmental facts and rule-conditioned decision information before policy optimization.

Encoders. To control the LLM input sequence length, a state encoder and a rule encoder encode the structured state and task rules into a small number of state tokens and rule tokens. These encoders

only transform the input format. The LLM still needs to learn the semantics of the state and rule information through Stage-A supervision.

Label Construction. For each environment, we fill a unified prompt template with the official environment description, task description, structured-state definition, and task rules. GPT-5.5 uses this information to generate the state-fact fields, rule-conditioned fields, and the corresponding executable Python annotation programs. After the fields are defined, the annotation programs generate labels for the collected data. State-fact labels are computed only from the structured state. Rule-conditioned labels are computed from both the structured state and the task rules. For selected rule-conditioned fields, we also pair the same structured state with rule variants that differ only in their numerical thresholds. We then regenerate the corresponding labels. This lets the model observe how the same state can receive different labels under different rule thresholds. Once the field definitions and task rules are fixed, the annotation programs deterministically generate the Stage-A supervision labels for each sample. All environments use the same field construction method and annotation procedure. The complete prompt template and implementation details are provided in the appendix.

3.3 Stage-A Supervision Objective

The Stage-A objective consists of a state-fact loss and a rule-conditioned loss. Given a batch of B samples, let $\mathcal{S}$ denote the set of state-fact fields and h_i the decision representation produced for sample i. For each field $k \in \mathcal{S}$, the corresponding readout head g_k^{state} predicts the state fact from h_i. The state-fact loss averages the classification losses over all state-fact fields within each sample and then across the batch:

$$\mathcal{L}_{\text{obs}} = \frac{1}{B|\mathcal{S}|} \sum_{i=1}^{B} \sum_{k \in \mathcal{S}} \text{CE}\big(g_k^{\text{state}}(h_i), y_{i,k}^{\text{state}}\big) . \tag{1}$$

$y_{i,k}^{\text{state}}$ is the deterministic supervision label for state-fact field k of sample i.

Rule-conditioned supervision is also implemented through field classification. Let $\mathcal{D}$ denote the set of rule-conditioned fields. For each field $j \in \mathcal{D}$, the readout head g_j^{dec} predicts its label from h_i. $y_{i,j}^{\text{dec}}$ is the supervision label for rule-conditioned field j of sample i, determined by the structured state and task rules. The rule-conditioned loss averages the classification losses over all rule-conditioned fields within each sample and then performs a weighted average across the batch:

$$\mathcal{L}_{\text{dec}} = \frac{1}{|\mathcal{D}| \sum_{i=1}^{B} w_i} \sum_{i=1}^{B} w_i \sum_{j \in \mathcal{D}} \text{CE}\big(g_j^{\text{dec}}(h_i), y_{i,j}^{\text{dec}}\big) . \tag{2}$$

The weight w_i is fixed during data construction; detailed sampling ratios and weight settings are provided in the appendix.

Finally, the Stage-A loss equally combines the state-fact loss and the rule-conditioned loss:

$$\mathcal{L}_{\text{StageA}} = \frac{1}{2} \left(\mathcal{L}_{\text{obs}} + \mathcal{L}_{\text{dec}}\right) . \tag{3}$$

Stage-A updates LoRA, the state and rule encoders, and the corresponding readout heads, while the Qwen base parameters and policy head remain frozen. The policy head is learned in Stage-B.

3.4 Stage-B: Online Policy Optimization with PPO

Stage-B uses the decision representation learned in Stage-A for online policy optimization. Stage-A encodes the environmental facts and the current decision objective into h_t, while Stage-B optimizes the policy using this representation. Stage-B inherits the Stage-A parameters used to form h_t and uses the same structured state and task rules as input to the LLM. The policy head (actor) takes h_t as input and produces a probability distribution over the native environment actions. The agent samples an action from this distribution, executes it in the environment, and receives the environment reward.

The critic receives only the history sequence as input, which we denote by x_t. We optimize the policy with PPO (Schulman et al., 2017). Let $\pi_\theta(a_t \mid h_t)$ denote the current policy and $\pi_{\theta_{\text{old}}}(a_t \mid h_t)$ the policy used to collect the trajectory. Their probability ratio is

$$\rho_t(\theta) = \frac{\pi_\theta(a_t \mid h_t)}{\pi_{\theta_{\text{old}}}(a_t \mid h_t)}. \tag{4}$$

We estimate the advantage using the critic. Let $V_\psi(x_t)$ be the value predicted by the critic. The temporal-difference residual is

$$\delta_t = r_t + \gamma V_\psi(x_{t+1}) - V_\psi(x_t), \tag{5}$$

and the generalized advantage estimate (Schulman et al., 2016) is

$$\hat{A}_t = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l}. \tag{6}$$

The PPO clipped policy objective is

$$\mathcal{L}_{\text{clip}} = \mathbb{E}_t \left[\min \left(\rho_t(\theta)\hat{A}_t, \text{clip}\left(\rho_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right]. \tag{7}$$

The critic is trained with the clipped value loss

$$\begin{aligned} V_t^{\text{clip}} &= V_{\psi_{\text{old}}}(x_t) + \text{clip}(V_\psi(x_t) - V_{\psi_{\text{old}}}(x_t), -\epsilon, \epsilon), \\ \mathcal{L}_{\text{value}} &= \frac{1}{2}\,\mathbb{E}_t \left[\max \left(\left(V_\psi(x_t) - \hat{R}_t \right)^2, \left(V_t^{\text{clip}} - \hat{R}_t \right)^2 \right) \right]. \end{aligned} \tag{8}$$

where $V_{\psi_{\text{old}}}(x_t)$ is the value estimate used when collecting the rollout, $\hat{R}_t$ is the return target, and ϵ is the PPO clipping coefficient. We also use the policy entropy

$$\mathcal{H} = \mathbb{E}_t \left[\mathcal{H}\big(\pi_\theta(\cdot \mid h_t)\big) \right] \tag{9}$$

to encourage exploration. The Stage-B optimization loss is

$$\mathcal{L}_{\text{StageB}} = -\mathcal{L}_{\text{clip}} + c_v \mathcal{L}_{\text{value}} - c_e \mathcal{H}. \tag{10}$$

During Stage-B, the Stage-A readout heads remain frozen and no longer participate in training. The Qwen base parameters also remain frozen. PPO updates the LoRA parameters, the state encoder, the rule encoder, the policy head, and the critic. The decision representation can therefore continue to adapt during policy optimization, while the field-level supervision heads are used only in Stage-A. Further PPO hyperparameters and implementation details are provided in the appendix.

4 Experimental Setup

Environments and Evaluation. We evaluate on five Atari games (SeaQuest, Alien, Assault, BeamRider, and DemonAttack) and Crafter (Bellemare et al., 2013; Hafner, 2022). The main experiments use approximately 1M Stage-B online environment interactions and 3 random seeds, reporting mean $\pm$ standard deviation. Atari results use the final-checkpoint game score and mean/median human-normalized score (HNS), while Crafter uses the official score. All methods are compared under the same Stage-B online interaction budget and evaluation protocol. Our method additionally uses pre-collected Stage-A supervision data, while all methods use the same online interaction budget during policy optimization. To evaluate cross-environment applicability, we also evaluate our method on the Sokoban and FrozenLake tasks used in RAGEN-2 (Wang et al., 2026) with Qwen2.5-3B and report success rate.

Training Configuration. For the Atari and Crafter experiments, all LLM-based methods except LaRe use Qwen3-1.7B (Yang et al., 2025a). Stage-A uses 200K supervised samples per Atari game and 61,902 samples from 100 official human episodes for Crafter. For LaRe, we use DeepSeek-V4-Flash only to generate the latent-factor evaluation function offline because Qwen3-1.7B does not

reliably produce executable functions under the same generation procedure. DeepSeek-V4-Flash is not involved in online reinforcement learning.

Baselines. We compare against PPO (Schulman et al., 2017), LLM-Explorer (Hao et al., 2025), LaRe (Qu et al., 2025), and Agent Learning via Early Experience (SR) (Zhang et al., 2026) to examine how different methods use LLMs to acquire task-relevant information and support subsequent policy learning. PPO uses the same policy model, structured-state and task-rule inputs, online interaction budget, and online training configuration as our method, but does not include Stage-A. All remaining details are provided in the appendix.

5 RESULTS

Table 1: Main experimental results under 1M online environment interactions and three random seeds. Atari scores are obtained by evaluating the final checkpoint of each seed over 100 episodes and are reported as mean ± standard deviation; HNS summarizes overall performance across games, while Crafter scores are computed using the official evaluation protocol.

Environment	Ours	PPO	LLM-Explorer	SR	LaRe
Atari					
SeaQuest	$\mathbf{1618.6 \pm 113.7}$	621.13 ± 31.00	222.2 ± 8.5	467.60 ± 35.08	196.87 ± 14.49
Alien	$\mathbf{1362.0 \pm 157.2}$	80.0 ± 86.60	287.9 ± 19.4	793.35 ± 187.88	709.53 ± 277.66
Assault	$\mathbf{5188.8 \pm 807.5}$	623.98 ± 9.85	277.8 ± 15.8	491.28 ± 74.40	715.96 ± 67.79
BeamRider	$\mathbf{5182.91 \pm 1217.25}$	246.0 ± 77.8	181.9 ± 9.3	570.99 ± 112.47	633.43 ± 66.87
DemonAttack	$\mathbf{8112.3 \pm 758.4}$	502.1 ± 27.9	728.5 ± 38.5	279.28 ± 150.78	1871.23 ± 631.61
Mean HNS	$\mathbf{288.54\%}$	19.00%	8.50%	13.83%	39.68%
Median HNS	$\mathbf{29.10\%}$	1.32%	0.87%	6.99%	6.98%
Crafter	$\mathbf{21.99 \pm 0.96}$	4.82 ± 0.43	1.08 ± 0.07	3.51 ± 0.19	2.612 ± 0.138

As shown in Table 1, our method outperforms all compared baselines on all five Atari games and Crafter under the same 1M online interaction budget. On Atari, our method achieves a mean HNS of 288.54% and a median HNS of 29.10%, compared with 39.68% and 6.98% for LaRe; on Crafter, it achieves 21.99, compared with 4.82 for the best baseline, PPO. These results show that learning decision-relevant domain knowledge before policy optimization leads to better policy performance.

Table 2: Success rate (%) on RAGEN-2 environments (Wang et al., 2026).

Method	Model	Sokoban	FrozenLake
RAGEN-2 (reported)	Qwen2.5-3B	28.90	77.90
PPO	Qwen2.5-3B	36.91	84.18
Ours	**Qwen2.5-3B**	**79.10**	**88.67**

Our method outperforms PPO on both Sokoban and FrozenLake, supporting its applicability beyond Atari and Crafter. The reported RAGEN-2 results are included for reference.

5.1 FIELD PREDICTION ACCURACY ANALYSIS

To verify whether Stage-A enables the LLM to learn decision-relevant domain knowledge, we compare how accurately the supervised fields can be predicted from h_t before and after Stage-A training. These fields represent the environmental facts in the structured state and the current decision objective determined from these facts and the task rules. We expect the supervised fields to be predicted more accurately from h_t after Stage-A training than before training. A large improvement indicates that Stage-A makes the decision-relevant domain knowledge recoverable from h_t.

However, Stage-A changes the latent representation space of h_t, and the readout heads are jointly trained with the Stage-A representation. Directly applying these heads to the pre-Stage-A h_t may

therefore cause the prediction accuracy to be affected by differences in the latent coordinate system. To reduce this effect, we align the pre-Stage-A representation space with the Stage-A representation space and then evaluate both representations using the same frozen Stage-A readout heads.

For each paired sample, we feed the same input to the LLM before and after Stage-A training and obtain h_t^{pre} and h_t^A. We consider two mappings. Orthogonal alignment accounts for translation, rotation, and reflection while preserving the geometric structure of the representation space. We also fit an unconstrained affine transformation to test whether the difference can be explained by a more general linear reparameterization. For both mappings, we use 10K, 20K, and 30K paired samples for fitting and evaluate on 2,000 unseen samples. The detailed formulations and optimization procedures are provided in the appendix.

Table 3: Latent-coordinate alignment on SeaQuest and Assault. All-field exact accuracy requires all supervised state-fact and rule-conditioned fields of a test sample to be predicted correctly.

Representation	**SeaQuest (15 fields)**	**Assault (16 fields)**
Pre-Stage-A	0.05%	1.65%
Orthogonal (10K)	2.10%	3.55%
Orthogonal (20K)	1.70%	3.70%
Orthogonal (30K)	1.85%	3.75%
Affine (10K)	17.75%	34.60%
Affine (20K)	21.15%	38.40%
Affine (30K)	21.25%	39.75%
Stage-A	**92.20%**	**97.20%**

The results in Table 3 show that, after Stage-A training, the all-field exact accuracy reaches 92.20% on SeaQuest and 97.20% on Assault, compared with only 0.05% and 1.65% before Stage-A training. Even after mapping the pre-Stage-A h_t using orthogonal alignment or a more general affine transformation, the highest accuracies remain only 21.25% and 39.75%, respectively, leaving a substantial gap from the post-Stage-A results. After Stage-A training, the environmental facts in the structured state and the current decision objective can be accurately recovered from h_t. This indicates that Stage-A encodes decision-relevant domain knowledge into the representation.

5.2 Ablation Studies

Table 4: Ablation results (mean ± standard deviation over three seeds). Atari scores average the final 100 training episodes; Crafter uses the official score computed over all training episodes. All variants within each environment use the same metric. Higher is better.

Environment	**Full Method**	**No Stage-A**	**No Rules**	**Single-Stage Training**	**Random Label**
SeaQuest	**1339.77±47.66**	479.0±28.1	1112.9±92.7	753.9±45.9	538.2±73.7
Assault	**3679.68±501.91**	564.6±105.2	2455.63±274.93	1938.8±269.7	1855.8±789.8
Alien	**1520.70±13.11**	298.8±30.3	1315.1±177.7	1191.9±42.9	656.4±26.9
Crafter	**21.99±0.96**	4.82±0.43	19.46±0.99	13.66±0.90	6.47±1.24

Ablation Studies. We conduct ablation studies to examine the roles of different components in learning decision-relevant domain knowledge before policy optimization.

No Stage-A. We completely remove Stage-A training while keeping all other settings unchanged. This variant exhibits a substantial performance drop across all evaluated environments. This shows that directly optimizing the policy with structured states and task rules does not achieve the same performance as first learning decision-relevant domain knowledge through Stage-A and then performing policy learning. **Learning this decision-relevant domain knowledge before policy optimization provides a more effective decision representation for subsequent policy learning and improves policy performance.**

Random Label. We preserve the Stage-A training procedure but randomly permute the supervision labels across samples, thereby breaking the correspondence between the inputs and the correct supervision targets. Its substantially lower performance than the full method shows that the gains from Stage-A arise from learning the correct state-fact and rule-conditioned supervision targets, rather than merely from introducing an additional stage of supervised training.

No Rules. We retain the structured-state inputs and state-fact supervision, but remove the task-rule inputs and rule-conditioned supervision. Its lower performance across all evaluated environments shows that learning environmental facts alone is insufficient, and that learning the current decision objective based on these facts and the task rules provides additional benefits.

Single-Stage Training. While keeping the total training budget of Stage-A and Stage-B unchanged, we jointly optimize the Stage-A supervision objectives and PPO within a single training stage. Its lower performance than the original two-stage training scheme shows that learning decision-relevant domain knowledge before policy optimization is more effective than learning this knowledge concurrently with policy optimization.

6 Conclusion

We study how to equip pretrained LLMs with decision-relevant domain knowledge before policy optimization in domain-specific reinforcement learning tasks. This knowledge includes the environmental facts in structured states and the current decision objective inferred from these facts and the task rules. We propose a two-stage framework that separates decision-relevant domain knowledge learning from policy optimization. In Stage-A, state-fact and rule-conditioned supervision train the LLM to encode this knowledge into a decision representation, while Stage-B inherits the Stage-A parameters used to form this representation and optimizes the policy with online PPO. Under the same online interaction budget, our method outperforms the compared baselines overall on Atari and Crafter and shows cross-environment applicability in additional environments. Representation analysis shows that, after Stage-A, the environmental facts and the current decision objective can be accurately recovered from the decision representation. Ablation studies further show the importance of correct state-fact and rule-conditioned supervision and of learning this knowledge before policy optimization. Overall, these results show that learning decision-relevant domain knowledge before policy optimization produces a decision representation that better supports subsequent policy learning and improves policy performance.

Limitations. A main limitation of our method is its reliance on structured-state definitions and task rules for constructing Stage-A supervision. Although the resulting annotation programs are deterministic once generated, incomplete or incorrect specifications may produce inaccurate supervision and cause the LLM to learn incorrect domain knowledge, which can in turn affect subsequent policy learning. This issue may become more pronounced in real-world tasks where state semantics and rules are incomplete, noisy, or difficult to specify. In addition, adapting the framework to a new environment still requires constructing and validating the corresponding supervision fields and annotation programs, which adds task-specific setup cost.

AI USE STATEMENT

In this work, we used generative AI tools to assist with research ideation and execution, including the construction of Stage-A supervision fields and executable annotation programs, as described in the paper. Generative AI tools were also used to support code development and debugging, literature search and related-work discovery, reference formatting, figure preparation, and the drafting and language polishing of parts of the manuscript. All AI-assisted outputs used in the method, experiments, and manuscript were reviewed and verified by the authors. In particular, the generated annotation programs were inspected and tested before being used to produce supervision labels, and experimental results and claims were independently checked by the authors. We take responsibility for the final content of this work, including text, claims, code, and artifacts produced with the aid of generative AI.

REFERENCES

Guillaume Alain and Yoshua Bengio. Understanding intermediate layers using linear classifier probes. In *International Conference on Learning Representations Workshop*, 2017.

Hao Bai, Yifei Zhou, Li Erran Li, Sergey Levine, and Aviral Kumar. Digi-Q: Learning VLM Q-value functions for training device-control agents. In *International Conference on Learning Representations*, 2025.

Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013. doi: 10.1613/jair.3912.

Daixuan Cheng, Shaohan Huang, and Furu Wei. Adapting large language models via reading comprehension. In *International Conference on Learning Representations*, 2024.

Younwoo Choi, Muhammad Adil Asif, Ziwen Han, John Willes, and Rahul G. Krishnan. Teaching LLMs how to learn with contextual fine-tuning. In *International Conference on Learning Representations*, 2025.

Quentin Delfosse, Jannis Bl"uml, Bjarne Gregori, Sebastian Sztwiertnia, and Kristian Kersting. OCAtari: Object-centric Atari 2600 reinforcement learning environments. *Reinforcement Learning Journal*, 1:400–449, 2024.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10764–10799. PMLR, 2023.

Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. Don't stop pretraining: Adapt language models to domains and tasks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 8342–8360. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.acl-main.740.

Danijar Hafner. Benchmarking the spectrum of agent capabilities. In *International Conference on Learning Representations*, 2022.

Qianyue Hao, Yiwen Song, Qingmin Liao, Jian Yuan, and Yong Li. LLM-Explorer: A plug-in reinforcement learning policy exploration enhancement driven by large language models. In *Advances in Neural Information Processing Systems*, volume 38, pp. 103786–103827, 2025. doi: 10.52202/085713-3129.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, pp. 3207–3214, 2018. doi: 10.1609/aaai.v32i1.11694.

John Hewitt and Percy Liang. Designing and interpreting probes with control tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pp. 2733–2743. Association for Computational Linguistics, 2019. doi: 10.18653/v1/D19-1275.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G. M. Araújo. CleanRL: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022. URL https://jmlr.org/papers/v23/21-1342.html.

Jinhao Jiang, Junyi Li, Wayne Xin Zhao, Yang Song, Tao Zhang, and Ji-Rong Wen. Mix-CPT: A domain adaptation framework via decoupling knowledge learning and format alignment. In *International Conference on Learning Representations*, 2025.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. Similarity of neural network representations revisited. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 3519–3529. PMLR, 2019.

Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, and Zhuowen Tu. Deeply-supervised nets. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, volume 38 of *Proceedings of Machine Learning Research*, pp. 562–570. PMLR, 2015.

Jiacheng Lin, Zhongruo Wang, Kun Qian, Tian Wang, Arvind Srinivasan, Hansi Zeng, Ruochen Jiao, Xie Zhou, Jiri Gesi, Dakuo Wang, Yufan Guo, Kai Zhong, Weiqi Zhang, Sujay Sanghavi, Changyou Chen, Hyokun Yun, and Lihong Li. SFT doesn't always hurt general capabilities: Revisiting domain-specific fine-tuning in LLMs. In *International Conference on Learning Representations*, 2026.

Jiahong Liu, Wenhao Yu, Quanyu Dai, Zhongyang Li, Jieming Zhu, Menglin Yang, Tat-Seng Chua, and Irwin King. PerFit: Exploring personalization shifts in representation space of LLMs. In *International Conference on Learning Representations*, 2026.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

Yecheng Jason Ma, Shagun Sodhani, Dinesh Jayaraman, Osbert Bastani, Vikash Kumar, and Amy Zhang. VIP: Towards universal visual reward and representation via value-implicit pre-training. In *International Conference on Learning Representations*, 2023.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Jim Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *International Conference on Learning Representations*, 2024.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015. doi: 10.1038/nature14236.

Ari S. Morcos, Maithra Raghu, and Samy Bengio. Insights on representational similarity in neural networks with canonical correlation. In *Advances in Neural Information Processing Systems*, volume 31, 2018.

Suraj Nair, Aravind Rajeswaran, Vikash Kumar, Chelsea Finn, and Abhinav Gupta. R3M: A universal visual representation for robot manipulation. In *Proceedings of The 6th Conference on Robot Learning*, volume 205, pp. 892–909. PMLR, 2023.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pp. 1310–1318. PMLR, 2013.

Yun Qu, Yuhang Jiang, Boyuan Wang, Yixiu Mao, Cheems Wang, Chang Liu, and Xiangyang Ji. Latent reward: LLM-empowered credit assignment in episodic reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 20095–20103, 2025. doi: 10.1609/aaai.v39i19.34213.

Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. SVCCA: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. In *Advances in Neural Information Processing Systems*, volume 30, pp. 6076–6085, 2017.

Peter H. Schönemann. A generalized solution of the orthogonal procrustes problem. *Psychometrika*, 31(1):1–10, 1966. doi: 10.1007/BF02289451.

John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations*, 2016.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

Hikaru Shindo, Quentin Delfosse, Devendra Singh Dhami, and Kristian Kersting. BlendRL: A framework for merging symbolic and neural policy learning. In *International Conference on Learning Representations*, 2025.

Zihan Wang, Chi Gui, Xing Jin, Qineng Wang, Licheng Liu, Kangrui Wang, Shiqi Chen, Linjie Li, Zhengyuan Yang, Pingyue Zhang, Yiping Lu, Jiajun Wu, Li Fei-Fei, Lijuan Wang, Yejin Choi, and Manling Li. RAGEN-2: Reasoning collapse in agentic RL. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026.

Zhengxuan Wu, Aryaman Arora, Zheng Wang, Atticus Geiger, Dan Jurafsky, Christopher D. Manning, and Christopher Potts. ReFT: Representation finetuning for language models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 63908–63962, 2024. doi: 10.52202/079017-2041.

Tianbao Xie, Siheng Zhao, Chen Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2reward: Reward shaping with language models for reinforcement learning. In *International Conference on Learning Representations*, 2024.

Kai Xiong, Xiao Ding, Yixin Cao, Li Du, Jiahao Ying, Yang Zhao, Bing Qin, and Ting Liu. Diagnosing and remedying knowledge deficiencies in LLMs via label-free curricular meaningful learning. In *International Conference on Learning Representations*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a. doi: 10.48550/arXiv.2505.09388.

Zitong Yang, Neil Band, Shuangping Li, Emmanuel Candès, and Tatsunori Hashimoto. Synthetic continued pretraining. In *International Conference on Learning Representations*, 2025b.

Kai Zhang, Xiangchao Chen, Bo Liu, Tianci Xue, Zeyi Liao, Zhihan Liu, Xiyao Wang, Yuting Ning, Zhaorun Chen, Xiaohan Fu, Jian Xie, Yuxuan Sun, Boyu Gou, Qi Qi, Zihang Meng, Jianwei Yang, Ning Zhang, Xian Li, Ashish Shah, Dat Huynh, Hengduo Li, Zi Yang, Sara Cao, Lawrence Jang, Shuyan Zhou, Jiacheng Zhu, Huan Sun, Jason Weston, Yu Su, and Yifan Wu. Agent learning via early experience. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*. PMLR, 2026. URL https://openreview.net/forum?id=N3dXUHY5dD.

A Additional Details

A.1 Structured States

For Atari environments, we construct structured states based on the object-centric representation provided by OCAtari. For other environments that expose native symbolic states, we follow the same information-construction principles and directly construct structured states from the native states exposed by the official environments.

Based on this, structured-state construction across all environments follows the following unified principles.

1. Preserve environment-provided entity categories, presence, and counts. The structured state first records which entities are contained in the current environment observation, whether they are currently present, and how many instances of each entity category are observed. For example, in SeaQuest, OCAtari (Delfosse et al., 2024) directly identifies objects such as Player, Shark, Submarine, Diver, and EnemyMissile. We retain their entity categories and record which objects are currently present together with the corresponding object counts. In Sokoban, Player, Box, Target, and Wall are directly obtained from the environment-provided `room_state` and `room_fixed`; FrozenLake similarly provides Player, Start, Goal, and Hole. In Crafter, this information corresponds to environmental entities within the player's observable neighborhood, such as Tree, Stone, Water, Cow, and Zombie.

2. Preserve environment-provided entity attributes and agent states. We further retain directly available entity attributes and the agent's own state, including position, orientation, resource values, survival status, and inventory information. For example, SeaQuest retains object positions, orientations, and oxygen level; Crafter retains the player's position, facing direction, health, food, drink, energy, and inventory items; Sokoban and FrozenLake retain the positions of the player and key map entities. These quantities describe objective properties of the environment state and do not incorporate judgments such as danger level, resource sufficiency, or action selection that require task-rule reasoning.

3. Allow deterministic transformations of existing environment information. Some raw environment information is complete but inconvenient to use directly. We therefore allow fixed deterministic transformations that make state representations easier to process consistently across environments. For example, absolute object coordinates may be converted into player-relative coordinates,

$$(x, y) \rightarrow (\Delta x, \Delta y),$$

and coordinates may be normalized according to the map or screen dimensions. Deterministic quantities such as object counts or geometric distances may also be computed from the observed state.

For example, relative positions can be derived from the player and object coordinates in SeaQuest; map coordinates in Sokoban and FrozenLake can be transformed into normalized and player-relative coordinates; and in Crafter, the player's world position is normalized while the locally observable region is organized into a fixed local grid representation.

All such transformations are uniquely determined by observed environment information and do not introduce task-rule judgments or policy-related information.

4. Use fixed-size deterministic representations when the number of entities varies. Some environments may contain a varying number of entities at a given time, whereas the model requires

a fixed-size input. We therefore organize these entities into a fixed number of object slots or fixed spatial slots using only the structural or geometric properties of the environment.

For example, SeaQuest follows the fixed object-slot organization used by the BlendRL representation (Shindo et al., 2025), where each object category has a predefined maximum number of slots. In Assault, BeamRider, and DemonAttack, when the number of same-category objects exceeds the available slots, objects are ordered according to their geometric distance from the player and the nearest objects are retained. Sokoban and FrozenLake instead use fixed map cells as spatial slots, such that slot definitions do not change with entity positions.

Overall, the structured state contains only information that can be deterministically obtained from the current or previously observed environment states.

Representative state examples. To make the representation concrete without exposing it as an opaque flat tensor, we show one real state from SeaQuest and one from Crafter in a human-readable grouped form. The values below are taken directly from the corresponding training data.

Table 5: One representative SeaQuest structured state. The underlying representation allocates 43 fixed object slots. All active slots in this frame are listed below; unused preallocated slots are zero-filled. For ordinary objects, a slot stores $[\text{present}, x, y, \text{orientation}]$. The OxygenBar uses its second field for the oxygen value.

Entity category	**Current content**
Player	1 object at $(86.0, 146.5)$, orientation code 12
Shark	4 objects at $(163.0, 143.5)$, $(4.0, 119.5)$, $(163.0, 95.5)$, and $(163.0, 71.5)$; orientation code 0
Submarine	No active object
Diver	4 objects at $(136.0, 74.5)$, $(136.0, 98.5)$, $(32.0, 122.5)$, and $(136.0, 146.5)$; orientation code 0
EnemyMissile	No active object
SurfaceSubmarine	No active object
PlayerMissile	1 object at $(69.0, 148.5)$, orientation code 0
OxygenBar	Oxygen value 60; stored as $[1, 60, 0, 0]$
CollectedDiver	5 objects at $(62.0, 182.5)$, $(70.0, 182.5)$, $(78.0, 182.5)$, $(86.0, 182.5)$, and $(94.0, 182.5)$

Table 6: One representative Crafter structured state shown by semantic component. The actual state is a 130-D vector. The local spatial component is only a 9×7 observable crop centered on the player, not the full world map; the fixed player cell is omitted, leaving 62 surrounding spatial entries. We summarize the salient contents here instead of printing all 62 cell values.

Component	Current content
Agent and resources	normalized position $(0.3333, 0.3650)$; facing down; health 8; food 6; drink 0; energy 9; sleeping 0; daylight 0.158; valid state 1
Inventory	stone $= 1$, coal $= 1$, wood pickaxe $= 1$, stone sword $= 1$; all other inventory entries are 0
Completed achievements	collect coal, collect stone, collect wood, eat cow, make stone sword, make wood pickaxe, place stone, place table, wake up; all remaining achievement flags are 0
Local observable crop	the full encoder retains all 62 surrounding cells. In this sample, the salient non-background contents are Zombie $\times 5$, Skeleton $\times 3$, Cow $\times 1$, Coal $\times 1$, Table $\times 1$, and Arrow $\times 1$; terrain cells (e.g., path, stone, grass, sand) are retained but omitted here for readability
Bounded observation history	A predefined number of history slots is used to retain previously observed facilities and environment objects, together with their player-relative positions and observation ages. This information is derived solely from past environment observations (e.g., a previously observed Table and two Stone objects in this sample).

A.2 Stage-A Label Construction

Stage-A supervision labels are generated by a deterministic semantic labeler constructed separately for each environment. Specifically, the labeler is a deterministic Python program generated by GPT-5.5 from the official environment description, task description, structured-state definition, and task rules according to our prompt template. GPT-5.5 first uses the prompt template to design the supervision fields and their label definitions, and then generates the corresponding Python labeling program to deterministically annotate the collected dataset and produce the Stage-A supervision labels. This use of executable programs follows the broader idea of delegating deterministic computation to generated code in program-aided language-model methods (Gao et al., 2023).

The prompt explicitly divides the supervision labels into two categories: *state-fact* labels and *rule-conditioned* labels. State-fact labels are deterministically computed only from information contained in the structured state and do not depend on the task rules. Rule-conditioned labels are deterministically computed from both the structured state and the task rules. We also explicitly prohibit GPT-5.5 from accessing or inferring any information related to expert data from the available inputs, in order to prevent the generated labels from containing prior knowledge derived from expert behavior. The resulting label schema is frozen before implementation. The generated labeler is then used solely as a deterministic program to produce the Stage-A supervision targets.

Because all environments use the same construction procedure, reproducing every environment-specific prompt would be unnecessarily repetitive. We therefore provide the shared prompt template and one complete SeaQuest instantiation below. The other environments follow the same procedure by replacing the environment description, task description, structured-state definition, and task rules with their environment-specific counterparts.

Prompt template. The shared template used to construct the deterministic Stage-A semantic labeler is:

```
You are given:

[Environment Description]: the observable entities, dynamics, and
   environment interface.

[Task Description]: the task objective and evaluation-relevant
   goals.
```

```
[Structured-State Definition]: the definitions and semantics of
   all structured-state fields available before the current
   decision, including
observable history when provided.

[Task Rules]: natural-language conditions, thresholds,
   prerequisites, and priorities.

Your task is to automatically design and implement a
   deterministic Python semantic
labeler for Stage-A supervision.

1. DESIGN THE SEMANTIC SCHEMA


Create two groups of labels.

State-fact labels: Select a minimal set of task-relevant semantic
   facts that can be
deterministically computed from Structured State alone and are
   invariant to Rules. Cover
relevant status, entities/candidates, relations,
   resources/progress, and observable
history when available. Continuous values may be abstracted only
   using rule-independent
and deterministic operations.

Rule-conditioned labels: Select the minimal semantic predicates
   required to express how Rules
interpret the current Structured State. Cover all
   behavior-relevant rule clauses that
are computable before the decision, including conditions,
   thresholds,
safety/opportunity, prerequisites/readiness, semantic targets,
   progress, and high-level
goals or priorities.

For each label specify:

name | values/range | deterministic definition | dependencies

Prefer compact structured labels when they preserve all necessary
   semantic distinctions.
Avoid redundant or synonymous labels.

2. ENFORCE SEMANTIC VALIDITY



Every label must be deterministically computable from the
   declared inputs.

State-fact labels must not depend on Rules.

Never use or infer hidden state, future observations, future
   reward/return, ground-truth
answers, successful trajectories, expert/teacher behavior,
   policy/value outputs, planner
```

```
outputs, actions, action preferences, or action sets.

Do not invent thresholds, semantic mappings, or hidden
    relationships that are not
explicitly provided.

If a rule cannot be computed from the available inputs, do not
    approximate it using a
proxy. Record it in UNIMPLEMENTABLE_RULES together with the
    reason and the minimal
missing observable information required to implement it.

All behavior-relevant rule clauses must either affect at least
    one Rule-conditioned label or
be explicitly declared unimplementable.

3. AUDIT AND FREEZE


Before implementation, audit the schema for:

- State-fact rule invariance;

- rule coverage;

- observability and pre-decision causality;

- deterministic implementability;

- redundancy;

- hidden-information or action leakage.


Revise internally until the audit passes, then freeze the schema.

After freezing, do not add, remove, rename, merge, split, or
    change label semantics
during implementation.

4. IMPLEMENT THE LABELER


Return one self-contained executable Python module containing:

- FROZEN_SCHEMA;

- RULE_COVERAGE;

- UNIMPLEMENTABLE_RULES;

- INPUT_GAPS;

- validate_structured_state(state);

- build_state_labels(state) -> dict;

- build_state_rule_labels(state, rules=None) -> dict;
```

```
- label_record(record) -> dict;

- label_records(records);

- audit_schema_code_consistency();

- self_test().


The runtime labeler must be pure and deterministic. It may use
    only Structured State and
explicitly supplied rule constants/parameters. It must not invoke
    another model, access
the environment, query external resources, or use information
    unavailable before the
decision.

If Rules are fixed when the labeler is generated, explicitly
    stated constants may be
compiled into the program. If Rules vary across records, their
    deterministic parameters
or identifiers must be supplied through the declared input
    interface.

5. VERIFY THE IMPLEMENTATION


self_test() must check at least:

- valid and invalid inputs;

- relevant rule-condition boundaries;

- empty or variable-size observable structures when applicable;

- observable history/progress when applicable;

- State-fact invariance to Rules;

- positive and negative cases for computable Rule-conditioned
    predicates;

- schema/code consistency;

- deterministic repeated execution.


The implementation must verify that returned label keys exactly
    match the frozen schema
and that every Rule clause is either covered or explicitly
    unimplementable.

OUTPUT


Running `python labeler.py` must execute self_test() and succeed
    only if all audits and
tests pass.
```

```
Optimize for a minimal, sufficient, deterministic, observable,
    and rule-covered semantic
supervision program, not for reproducing any unknown
    human-designed label schema. When
information is insufficient, fail closed and report the missing
    dependency rather than
hallucinating it.

FINAL CODE-SAFETY REQUIREMENTS

All schema/coverage/gap metadata must be valid Python literals
    and JSON-serializable;
every descriptive phrase must be quoted as a string.

`label_record(record)` must return exactly `{"state_only":
    build_state_labels(...),
"state_rule": build_state_rule_labels(...)}`.

Any self-test using `label_record` must access labels through
    those two top-level keys,
or call the corresponding build function directly.

Before returning the module, perform a final internal
    syntax/interface audit: every test


reference must match the actual function return structure, every
    referenced label key
must exist in the frozen schema, and the complete module must be
    syntactically valid
Python.

FINAL ROBUSTNESS REQUIREMENTS

Assign each supplied Rule clause one canonical stable rule ID
    exactly once.
RULE_COVERAGE, UNIMPLEMENTABLE_RULES, INPUT_GAPS, and consistency
    audits must reference
those same IDs exactly; do not compare paraphrased rule text.

Keep the generated module concise: prefer reusable helper
    functions and compact fixtures
over repeated hand-written literals.

Before returning, re-check all quotes, brackets, commas, function
    return structures, and
test references so the full output is syntactically valid Python
    and internally
consistent.
```

Complete SeaQuest instantiation. The following is the complete SeaQuest prompt obtained by filling the template with the SeaQuest environment description, task description, structured-state definition, and task rules:

```
[Environment Description]

Environment: SeaQuest

[Task Description]
```

```
Task:
Control a submarine, survive hostile threats, manage oxygen,
rescue divers, deliver rescued divers, and accumulate reward.

Observable task dynamics:
- The player submarine moves underwater and can return to the
    surface.
- Hostile submarines, sharks, and enemy projectiles can threaten
    the player.
- Oxygen is a survival resource.
- Divers are observable collectible/rescue targets.
- Rescued-diver progress is observable.
- Survival, oxygen recovery, diver collection, diver delivery,
  and ordinary exploration can compete as high-level objectives.


[Structured-State Definition]

The pre-decision structured state provides enough deterministic
    information to compute:
- player position / region
- motion from observable history
- nearest threat type
- nearest threat relative side/direction
- nearest threat distance / collision relation
- oxygen level
- number of currently carried divers


- number of visible free divers
- nearest diver direction
- nearest diver distance
- safety of a diver collection opportunity
- whether six divers are currently carried
- observable state history when supplied

No future observation, future reward, teacher action, policy
    output,
action preference, or Action-Set is supplied.


[Task Rules]

R1. Survive before pursuing score.

R2. Use calibrated danger-threshold setting 0.50;
    evade when a same-lane threat crosses that threshold.

R3. Use calibrated oxygen-threshold setting 0.50;
    ascend when oxygen crosses that threshold outside the surface
    refill zone.

R4. Ascend to deliver when carrying 6 divers.

R5. Use calibrated diver-collection threshold setting 0.50;
    collect a diver inside that threshold only when safe.

R6. Priority:
```

```
    evade and ascend;
    evade;
    air;
    deliver;
    collect;
    explore.

R7. The rule constrains the goal but not one exact action.

1. DESIGN THE SEMANTIC SCHEMA


Create two groups of labels.


State-fact labels: Select a minimal set of task-relevant semantic
   facts that can be
deterministically computed from Structured State alone and are
   invariant to Rules. Cover
relevant status, entities/candidates, relations,
   resources/progress, and observable
history when available. Continuous values may be abstracted only
   using rule-independent
and deterministic operations.

Rule-conditioned labels: Select the minimal semantic predicates
   required to express how Rules
interpret the current Structured State. Cover all
   behavior-relevant rule clauses that
are computable before the decision, including conditions,
   thresholds,
safety/opportunity, prerequisites/readiness, semantic targets,
   progress, and high-level
goals or priorities.

For each label specify:

name | values/range | deterministic definition | dependencies

Prefer compact structured labels when they preserve all necessary
   semantic distinctions.
Avoid redundant or synonymous labels.

2. ENFORCE SEMANTIC VALIDITY



Every label must be deterministically computable from the
   declared inputs.

State-fact labels must not depend on Rules.

Never use or infer hidden state, future observations, future
   reward/return, ground-truth
answers, successful trajectories, expert/teacher behavior,
   policy/value outputs, planner
outputs, actions, action preferences, or action sets.
```

```
Do not invent thresholds, semantic mappings, or hidden
    relationships that are not
explicitly provided.

If a rule cannot be computed from the available inputs, do not
    approximate it using a
proxy. Record it in UNIMPLEMENTABLE_RULES together with the
    reason and the minimal
missing observable information required to implement it.

All behavior-relevant rule clauses must either affect at least
    one Rule-conditioned label or
be explicitly declared unimplementable.

3. AUDIT AND FREEZE


Before implementation, audit the schema for:

- State-fact rule invariance;

- rule coverage;

- observability and pre-decision causality;


- deterministic implementability;

- redundancy;

- hidden-information or action leakage.


Revise internally until the audit passes, then freeze the schema.

After freezing, do not add, remove, rename, merge, split, or
    change label semantics
during implementation.

4. IMPLEMENT THE LABELER


Return one self-contained executable Python module containing:

- FROZEN_SCHEMA;

- RULE_COVERAGE;

- UNIMPLEMENTABLE_RULES;

- INPUT_GAPS;

- validate_structured_state(state);

- build_state_labels(state) -> dict;

- build_state_rule_labels(state, rules=None) -> dict;

- label_record(record) -> dict;
```

```
- label_records(records);

- audit_schema_code_consistency();

- self_test().


The runtime labeler must be pure and deterministic. It may use
    only Structured State and
explicitly supplied rule constants/parameters. It must not invoke
    another model, access
the environment, query external resources, or use information
    unavailable before the
decision.



If Rules are fixed when the labeler is generated, explicitly
    stated constants may be
compiled into the program. If Rules vary across records, their
    deterministic parameters
or identifiers must be supplied through the declared input
    interface.

5. VERIFY THE IMPLEMENTATION


self_test() must check at least:

- valid and invalid inputs;

- relevant rule-condition boundaries;

- empty or variable-size observable structures when applicable;

- observable history/progress when applicable;

- State-fact invariance to Rules;

- positive and negative cases for computable Rule-conditioned
    predicates;

- schema/code consistency;

- deterministic repeated execution.


The implementation must verify that returned label keys exactly
    match the frozen schema
and that every Rule clause is either covered or explicitly
    unimplementable.

OUTPUT

Return only the complete Python module, without explanatory prose
    or Markdown fences.
```

```
Running `python labeler.py` must execute self_test() and succeed
    only if all audits and
tests pass.

Optimize for a minimal, sufficient, deterministic, observable,
    and rule-covered semantic
supervision program, not for reproducing any unknown
    human-designed label schema. When
information is insufficient, fail closed and report the missing
    dependency rather than
hallucinating it.

FINAL CODE-SAFETY REQUIREMENTS

All schema/coverage/gap metadata must be valid Python literals
    and JSON-serializable;
every descriptive phrase must be quoted as a string.

`label_record(record)` must return exactly `{"state_only":
    build_state_labels(...),
"state_rule": build_state_rule_labels(...)}`.

Any self-test using `label_record` must access labels through
    those two top-level keys,
or call the corresponding build function directly.

Before returning the module, perform a final internal
    syntax/interface audit: every test


reference must match the actual function return structure, every
    referenced label key
must exist in the frozen schema, and the complete module must be
    syntactically valid
Python.

FINAL ROBUSTNESS REQUIREMENTS

Assign each supplied Rule clause one canonical stable rule ID
    exactly once.
RULE_COVERAGE, UNIMPLEMENTABLE_RULES, INPUT_GAPS, and consistency
    audits must reference
those same IDs exactly; do not compare paraphrased rule text.

Keep the generated module concise: prefer reusable helper
    functions and compact fixtures
over repeated hand-written literals.

Before returning, re-check all quotes, brackets, commas, function
    return structures, and
test references so the full output is syntactically valid Python
    and internally
consistent.
```

A.3 Dataset Construction

For the Atari environments, we train PPO agents following the implementation and default training setup of CleanRL (Huang et al., 2022). After obtaining the final PPO checkpoints, we use the corresponding trained agent to collect 200K states for each Atari game. For Crafter (Hafner, 2022), we use 61,902 states from 100 official human trajectories as the training data. For Sokoban and FrozenLake, data are collected directly from the official environments by uniformly sampling environment actions.

All states listed in Table 7 are used for Stage-A training. For the latent-coordinate alignment analysis, the alignment mappings are fitted using 10K, 20K, or 30K samples drawn from the Stage-A training set, while evaluation uses 2,000 held-out states from the test split that do not overlap with Stage-A training. Stage-A is trained using a predefined training configuration, and the final checkpoint is directly used to initialize Stage-B training.

Table 7: Dataset sources and sizes used for Stage-A training.

Environment	**Data Source**	**Total States**	**Stage-A Training States**
SeaQuest	CleanRL PPO	200K	200K
Assault	CleanRL PPO	200K	200K
BeamRider	CleanRL PPO	200K	200K
DemonAttack	CleanRL PPO	200K	200K
Alien	CleanRL PPO	200K	200K
Crafter	100 official human trajectories	61,902	61,902
Sokoban	Official environment, uniform actions	50,000	50,000
FrozenLake	Official environment, uniform actions	6,000	6,000

A.4 Stage-A Training Details

Stage-A uses the pre-constructed supervision labels to provide task-related supervision for learning the decision representation h_t. During training, the base Qwen parameters and the downstream policy head remain frozen, while only the LoRA parameters (Hu et al., 2022), state and rule encoders, and the Stage-A readout heads are optimized. Lightweight linear readouts are widely used to expose information available in hidden representations (Alain & Bengio, 2017; Hewitt & Liang, 2019); related auxiliary-classification objectives have also been used to shape intermediate representations during training (Lee et al., 2015). Here, our readout heads are part of the Stage-A supervision objective rather than post-hoc diagnostic probes. The training objective follows the Stage-A supervision objective defined in the main text: classification losses are computed separately for the state-fact fields and the rule-conditioned fields and are combined with equal weights.

For Atari and Crafter, we use a logical batch size of 128. Each batch contains 64 natural samples, 38 samples in which rule conditions are triggered, and 26 paired samples formed from 13 identical states under different rules. Because paired samples are less frequent, they use a fixed sample weight of $102/26$ in the rule-conditioned decision loss to balance the contributions of non-paired and paired samples to the training objective. All environments use a microbatch size of 64, with gradient accumulation used to preserve the logical batch size of 128. We optimize with AdamW (Kingma & Ba, 2015; Loshchilov & Hutter, 2019), using linear warmup for the first 3% of the total training steps followed by linear learning-rate decay to zero, together with gradient clipping (Pascanu et al., 2013). The detailed training configuration is shown below.

Table 8: Stage-A training hyperparameters.

Stage-A setting	**Value**
Logical batch size	128
Microbatch size	64
Optimizer	AdamW
LoRA rank / α	32 / 64
LoRA dropout	0
LoRA learning rate	1×10^{-4}
Encoder learning rate	1×10^{-4}
Semantic readout-head learning rate	3×10^{-4}
Weight decay	0.01
Warmup ratio	0.03
Learning-rate schedule	Linear warmup + linear decay
Maximum gradient norm	1.0
State-fact / Rule-conditioned loss weight	0.5 / 0.5
Qwen base parameters	Frozen
Policy head	Frozen
All environments batch composition	64 natural + 38 triggered + 26 paired
Paired-sample weight	$102/26 \approx 3.923$
Atari / Crafter training budget	25,250 optimizer steps
Sokoban training budget	15,000 optimizer steps
FrozenLake training budget	2,000 optimizer steps

A.5 STAGE-B PPO DETAILS

Stage-B initializes the actor from the Stage-A checkpoint and performs pure online policy optimization with PPO (Schulman et al., 2017). The semantic readout heads introduced in Stage-A are frozen and no longer participate in the training objective, while the base Qwen parameters also remain frozen. The parameters optimized in Stage-B include the LoRA parameters, encoders, policy head, and a separately initialized value critic. At each decision step, the actor takes only the decision representation h_t as input. The critic receives only the history sequence.

We use the standard clipped PPO objective together with generalized advantage estimation (GAE) (Schulman et al., 2016), advantage normalization, a clipped value-function objective, and entropy regularization. Learning rates are linearly annealed during training, and gradient clipping is applied to improve optimization stability. For Atari and Crafter, each rollout uses 8 parallel environments with 128 steps collected per environment, resulting in a PPO batch of 1,024 transitions; the total training budget is 1,000,448 online environment interactions. For Sokoban and FrozenLake, we use the same setting described in RAGEN-2 (Wang et al., 2026): each update contains 8 environment groups, with 16 trajectories per group and at most 10 actions per trajectory, for a total of 400 PPO updates. Apart from this, all environments use the same PPO optimization hyperparameters shown below.

Table 9: Stage-B PPO optimization hyperparameters shared across environments.

Hyperparameter	Value
PPO epochs per update	1
Logical minibatch size	256
Microbatch size	64
Discount factor γ	0.99
GAE parameter λ	0.95
PPO clip coefficient	0.10
Target KL	0.02
Entropy coefficient	0.01
Value-loss coefficient	0.50
Maximum gradient norm	0.50
LoRA rank / α	32 / 64
LoRA learning rate	5×10^{-6}
Encoder learning rate	5×10^{-5}
Policy-head learning rate	2.5×10^{-4}
Value-head learning rate	2.5×10^{-4}
Critic hidden size	256
Optimizer ϵ	1×10^{-5}
Learning-rate schedule	Linear decay

A.6 Baseline Implementation Details

We reproduce or adapt each baseline from its released implementation while preserving its main training procedure and input interface. PPO uses the same policy model as our method without Stage-A, while LLM-Explorer, LaRe, and SR retain their own method-specific architectures and training procedures. For the main Atari experiments, we use the same approximately 1M online decision budget and three random seeds.

PPO. The PPO baseline uses the same policy model, structured-state and task-rule inputs, value critic, reward processing, and Stage-B PPO configuration as our method. It does not load a Stage-A checkpoint or perform any Stage-A optimization. The baseline therefore starts online PPO directly from the pretrained Qwen initialization together with newly initialized trainable modules. We use 1,000,448 online interactions and random seeds 0, 1, and 2.

LLM-Explorer. We implement LLM-Explorer (Hao et al., 2025) based on its released Atari code and preserve its DQN learning pipeline (Mnih et al., 2015), pixel preprocessing, experience replay, ϵ-greedy exploration, and LLM-guided exploration mechanism. The LLM summarizes recent interaction experience and produces an action distribution for subsequent exploration. To control the LLM scale across methods, we use Qwen3-1.7B with $M = 100$, $K = 1$, and temperature 1.0. We keep the Atari environment and reward-processing settings fixed across seeds. We also use a fixed game-independent safeguard for abnormal LLM-generated exploration distributions. No task-specific information is introduced by this safeguard.

LaRe. We adapt the official LaRe implementation (Qu et al., 2025) to Atari while preserving its latent-factor reward construction, reward decoder, and PPO training procedure. An LLM first generates an executable latent-factor evaluation function from the task information and structured-state specification. We use DeepSeek-V4-Flash only for this offline function-generation step because Qwen3-1.7B does not reliably generate executable functions under the same procedure. The generated function is then fixed and DeepSeek-V4-Flash does not participate in online reinforcement learning. We use the corresponding Atari structured states as the observation interface and match the online interaction budget to our setting.

SR. We adapt the released Self-Reflection pipeline from Agent Learning via Early Experience (Zhang et al., 2026) to Atari while preserving its reflection and supervised fine-tuning procedure. We select 5,000 states from high-return PPO-teacher trajectories. At each state, the Qwen3-1.7B policy proposes $K = 3$ non-expert alternative actions. We restore the same Atari emulator state and execute the expert action and each alternative separately to obtain their actual next-state

outcomes. These outcomes are used to generate reflection data without providing reward information to the reflection model. We combine the resulting reflection data with the expert data for supervised fine-tuning. The resulting policy is then optimized with PPO under the same approximately 1M online interaction budget and three-seed evaluation protocol.

Crafter adaptation. For Crafter, we adapt LLM-Explorer, LaRe, and SR to the same Crafter environment and the native 17-action space. All three baselines use an online interaction budget of approximately 1M decisions and are evaluated with three random seeds. We keep the original core training procedure of each baseline and only adapt the environment interface when needed.

LLM-Explorer keeps its DQN training pipeline and LLM-guided exploration mechanism, with Qwen3-1.7B used as the LLM. LaRe keeps its LLM-generated latent-factor function, learned reward model, and subsequent PPO training. SR keeps its early-experience construction, reflection generation, supervised fine-tuning, and subsequent PPO training. We do not add our Stage-A supervision or other components specific to our method to these baselines.

All methods use the same Crafter evaluation protocol. The Crafter score (Hafner, 2022) is computed from the success rates of 22 achievements:

$$\text{Score} = \exp\left(\frac{1}{22}\sum_{i=1}^{22}\log(1+s_i)\right) - 1,$$

where s_i is the success rate of achievement i. We report the mean and standard deviation over three random seeds.

A.7 Latent-Coordinate Alignment Details

For each paired sample, we feed exactly the same $(\text{state}, \text{history}, \text{rule})$ input to the LLM before and after Stage-A training, obtaining the corresponding decision representations h_t^{pre} and h_t^A. Comparing learned representations across layers, models, or training stages has been studied using several alignment and similarity measures (Raghu et al., 2017; Morcos et al., 2018; Kornblith et al., 2019). For N samples, these representations form $X, Y \in \mathbb{R}^{N\times d}$. We first subtract the mean of each representation space,

$$X_c = X - \mu_{\text{pre}}, \qquad Y_c = Y - \mu_A, \tag{11}$$

so that the centers of the two point clouds are translated to the origin, removing their global translation difference. We then seek the orthogonal transformation that best aligns the two coordinate systems using the classical orthogonal Procrustes formulation (Schönemann, 1966),

$$R^* = \arg\min_R \|X_c R - Y_c\|_F^2, \qquad R^T R = I. \tag{12}$$

Performing singular value decomposition on

$$X_c^T Y_c = U\Sigma V^T \tag{13}$$

gives the optimal orthogonal matrix

$$R^* = UV^T. \tag{14}$$

Because R^* is orthogonal, it can only rotate or reflect the representation space and does not change geometric structures such as distances and angles. For a test sample, the aligned representation is

$$\tilde{h}_t^{\text{pre}} = (h_t^{\text{pre}} - \mu_{\text{pre}})R^* + \mu_A. \tag{15}$$

Orthogonal alignment only accounts for translation, rotation, and reflection. To further test whether the remaining difference can be explained by a more general linear reparameterization, we additionally fit an unconstrained affine transformation,

$$\tilde{h}_t^{\text{pre}} = h_t^{\text{pre}} A + b, \tag{16}$$

where A and b are obtained by ordinary least squares,

$$(A^*, b^*) = \arg\min_{A,b} \|XA + b - Y\|_F^2. \tag{17}$$

Without the orthogonality constraint, this transformation allows rotation, scaling, shearing, and arbitrary linear reorganization across dimensions. For both orthogonal alignment and affine transformation, we separately fit the mappings using 10K, 20K, and 30K paired samples and evaluate each fitted mapping on 2,000 unseen samples.

A.8 Ablation Implementation Details

Across all ablation variants, we keep the model architecture, Stage-B PPO configuration, online interaction budget, and ablation evaluation protocol unchanged unless otherwise specified. In **No Stage-A**, we completely skip supervised Stage-A training and directly optimize the policy with PPO. No Stage-A checkpoint is loaded, while the structured-state and task-rule inputs used by the policy remain available. In **Random Label**, we keep the Stage-A data, model structure, sampling procedure, loss weights, optimizer schedule, and Stage-B configuration unchanged, but replace the Stage-A supervision targets in the training split using a fixed label permutation. This permutation breaks the correspondence between each input and its correct semantic labels while preserving the original label spaces and sampling structure. Validation and test labels are not permuted.

In **No Rules**, we remove both the task-rule inputs and the rule-conditioned supervision in Stage-A, while retaining the structured-state inputs and state-fact supervision. The rule-input branch is also removed from the representation module, and the remaining training settings are kept unchanged. In **Single-Stage Training**, we keep the same Stage-A supervision budget and PPO interaction budget as in the two-stage method, but interleave the two optimization processes instead of completing Stage-A before policy optimization. Specifically, the experiment uses 25,250 semantic optimization steps and 1,000,448 online environment interactions, corresponding to 977 PPO updates. The semantic objective and PPO use separate optimizers, while the shared representation parameters can be updated by both objectives. We distribute the semantic updates throughout PPO training, with approximately 25–26 semantic updates between successive PPO updates. This design changes the training order of the two objectives while keeping their optimization budgets unchanged.

A.9 Compute and Reproducibility

Experiments are conducted on a workstation with two NVIDIA RTX 4090 GPUs with 24 GB memory each. Our main training environment uses Python 3.11.13, PyTorch 2.4.0 with CUDA 12.1, Transformers 4.51.3, PEFT 0.15.2, and NumPy 1.26.4. Because deep reinforcement learning results can vary substantially across independent runs (Henderson et al., 2018), main experiments use three random seeds, 0, 1, and 2. A single training run uses one GPU, while independent runs can be executed in parallel across the two GPUs. Based on the completed runs, Stage-A typically requires approximately 3–5.5 hours per environment, while Stage-B with approximately 1M online interactions requires approximately 3.3–5.5 hours per seed.

The main Atari results in Table 1 are obtained from the final Stage-B checkpoint after 1,000,448 online interactions. We do not select Stage-B checkpoints according to evaluation performance. For these results, each final checkpoint is evaluated on 100 full games using deterministic argmax actions, and the reported result is computed across seeds 0, 1, and 2.

A.10 ATARI TRAINING CURVES

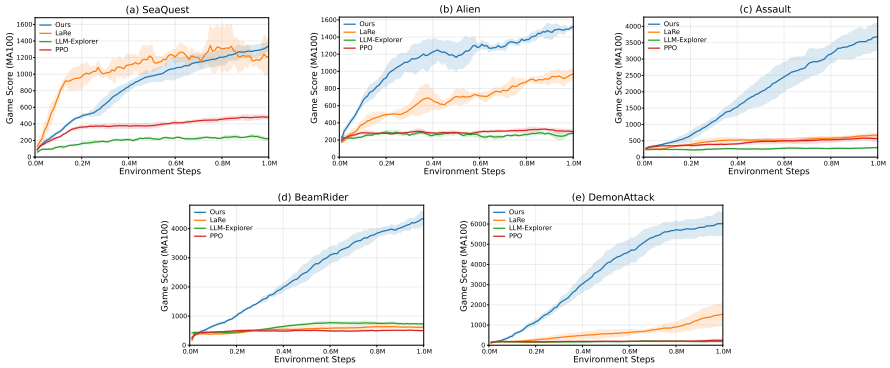


Figure 5: Atari training curves under approximately 1M online environment interactions. Curves show the mean across three random seeds with ± 1 standard deviation.