LEARNING AGENTIC VISION-LANGUAGE GAME AGENTS FROM TRAJECTORY-DERIVED SUPERVISION

Anonymous authors
Paper under double-blind review

ABSTRACT

Vision-language game agents are typically trained to predict the next action from gameplay demonstrations in a supervised fine-tuning (SFT) manner. While effective for learning short-horizon behavior, this training view underuses the process structure already present in trajectories, including how plans persist across multiple steps, how actions are carried out under a plan, and how execution outcomes signal when a new plan is needed. Gameplay trajectories contain richer supervision than action labels alone. When organized into plan-conditioned segments, they naturally specify how an agent should form a plan, act under that plan over time, and track whether execution is still aligned with it. In this paper, we propose a trajectory-to-process supervision framework for learning plan-act-verify vision-language game agents from gameplay trajectories. Unlike conventional CoT-before-action training, where plans mainly serve as intermediate prefixes for action prediction, our formulation explicitly trains the model to carry out a plan across multiple steps and to generate the next plan when the current one succeeds, fails, or is no longer valid. The resulting agent moves beyond direct acting toward closed-loop gameplay, using trajectory-derived process annotations as supervision for agentic behavior. Across Tetris, Sokoban, and MiniGrid, our method consistently improves over direct action prediction, step-wise plan-as-CoT, and open-loop plan-chunk baselines, and transfers across Qwen3-VL-8B, InternVL3-8B, and Gemma3-12B backbones.

1 INTRODUCTION

Vision-language game agents are often trained by treating gameplay demonstrations as collections of observation-action pairs. This training view can be effective for learning reactive action prediction, and recent vision-language game agents have shown that supervised fine-tuning on gameplay trajectories can produce competent policies (Tan et al., 2025a; Chen et al., 2025; Wang et al., 2025b). However, such a formulation does not fully exploit the sequential structure already present in trajectories. In many games, effective behavior depends not only on selecting an action from the current visual observation, but also on forming and sustaining temporally extended decisions across multiple steps. When trajectories are reduced to isolated observation-action pairs, these longer-horizon dependencies are only weakly reflected in the supervision signal, and the learned policy is optimized primarily for local action prediction.

Existing work addresses this limitation through richer policy supervision or explicit test-time workflows. The former augments action learning with explanations, reasoning traces, or intermediate plans (Tan et al., 2025a; Chen et al., 2025; Wang et al., 2025b; Zhao et al., 2025). The latter uses reflection or hierarchical planner-executor pipelines to organize execution (Zhang et al., 2026; Wang et al., 2025a; SIMA Team et al., 2025). However, treating a plan only as a per-step action prefix does not explicitly supervise its execution across subsequent steps (Kwok et al., 2026). This leaves a complementary training question: how can plan generation, multi-step execution, and plan transitions be learned jointly within a policy from target-game trajectories?

We propose a trajectory-to-process supervision framework that organizes annotated gameplay trajectories into plan-conditioned segments. Each segment links a local plan to the actions executed under it and execution-status labels indicating whether to continue or replan. Joint supervision trains a single vision-language policy to generate plans, execute them across multiple steps, and

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.

predict when they should be replaced. Our contribution is to make these relationships explicitly trainable from trajectories, extending supervision beyond isolated action labels or per-step reasoning prefixes. Experiments on Tetris, Sokoban, and MiniGrid show improved task performance over direct action prediction, step-wise plan-as-CoT, and open-loop plan-chunk baselines, together with stronger alignment between generated plans and realized behavior.

2 Related Work

Vision-language game agents. Recent vision-language game agents have gradually moved beyond direct action prediction by incorporating richer supervision into gameplay learning. Early trajectory-driven agents such as VPT (Baker et al., 2022) and STEVE-1 (Lifshitz et al., 2023) showed that gameplay trajectories can support visually grounded behavior and instruction following in Minecraft. Subsequent work has explored staged post-training (Li et al., 2025a), explanatory outputs (Tan et al., 2025a; Chen et al., 2025), mixed reasoning-and-action trajectories (Wang et al., 2025b), and trainable abstract actions (Wang et al., 2025a). These methods use additional signals to improve game understanding and action generation. Recent CoT-VLA methods similarly train models to generate intermediate reasoning before actions (Zhao et al., 2025; Zhong et al., 2026; Wang et al., 2026). Our work is related in that it also starts from gameplay trajectories, but differs by converting their temporal structure into process-level supervision for planning, plan-conditioned execution, and execution-status prediction.

Goal- and subgoal-conditioned imitation learning. Our formulation is also related to goal- and subgoal-conditioned imitation learning, where demonstrations are reused to train policies that act toward specified future states or intermediate objectives. Goal-conditioned imitation learning studies how demonstrations can accelerate the learning of policies that reach arbitrary goals, including settings with action-free or suboptimal demonstrations (Ding et al., 2019). Hierarchical imitation learning further decomposes long-horizon behavior into high-level subgoal selection and low-level control, reducing exploration and supervision costs in sparse-reward tasks (Le et al., 2018). In robotic manipulation, Learning Latent Plans from Play learns reusable latent plans from unsegmented play data and conditions policies on inferred goals and latent plans (Lynch et al., 2020), while Relay Policy Learning trains goal-conditioned hierarchical policies from unstructured demonstrations and then improves them with reinforcement learning (Gupta et al., 2020). These works show that conditioning imitation policies on goals, subgoals, or latent plans is effective for long-horizon control. Unlike prior goal-conditioned imitation methods that primarily assume a goal or subgoal as an input condition for policy learning, our framework learns a full plan-act-verify process: generating plans, acting under them, and predicting when to replan.

Agentic workflows for long-horizon decision making. Another line of work addresses long-horizon decision making with explicit agentic workflows. GameVerse (Zhang et al., 2026) studies reflect-and-retry using failed trajectories and tutorial videos as external feedback, while Cradle (Tan et al., 2025b), Voyager (Wang et al., 2024), and DEPS (Wang et al., 2023) organize planning, reflection, skill construction, and execution into multi-stage pipelines. Optimus-2 (Li et al., 2025b) likewise separates high-level multimodal planning from low-level action generation. These systems highlight the importance of explicit planning and revision during deployment. In contrast, our focus is on how such plan-act-verify behavior can be learned from trajectory-derived supervision within a single trainable vision-language game policy.

Execution monitoring and replanning. In embodied control, recent methods have placed increasing emphasis on execution monitoring after planning, especially through progress awareness, recovery, and replanning (Ichter et al., 2023). See, Plan, Rewind (Dai et al., 2026) and Agentic Robot (Yang et al., 2025) make this concern explicit through progress-aware or verifier-driven closed-loop execution, where monitored execution status is used to trigger recovery. PALM (Liu et al., 2026) brings the same concern into policy learning through explicit progress cues, while OneTwoVLA (Lin et al., 2025) and NovaPlan (Fu et al., 2026) extend it further to adaptive reasoning, replanning, and recovery as part of the execution process itself. These works show that robust long-horizon behavior requires tracking execution progress rather than only predicting the next action. Our formulation shares this emphasis, but studies it in a different setting: rather than introducing progress tracking or recovery through specialized robotic control designs, we study how gameplay trajectories can be

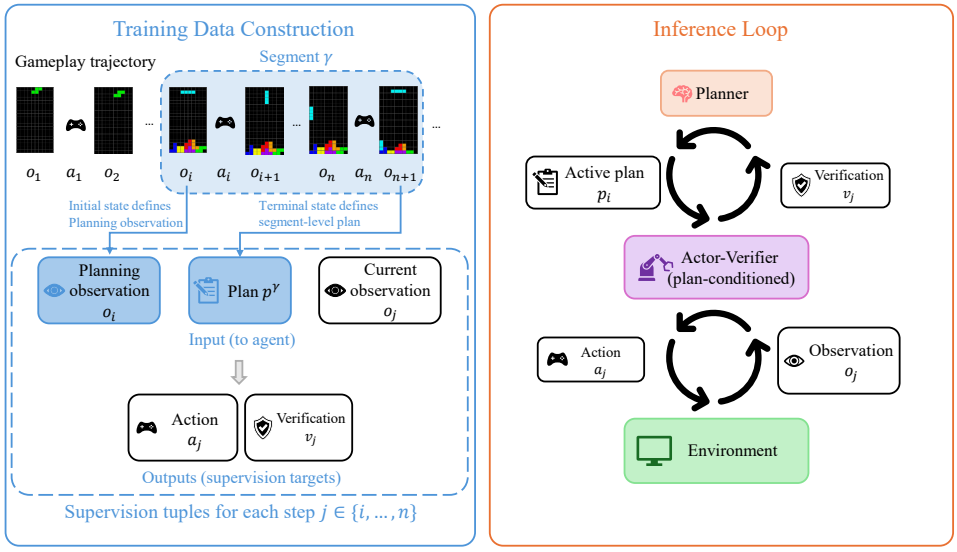


Figure 1: Overview of the proposed trajectory-to-process supervision framework. **Left:** trajectories are segmented into plan-conditioned segments, where the terminal state of a segment defines a segment-level plan that is assigned to all steps in the segment. **Right:** during online evaluation, the agent acts under an active plan and uses the predicted verification state to decide whether to continue or replan.

converted into supervision for plan-conditioned execution and execution-status prediction within a single trainable game policy.

3 METHOD

In this section, we introduce our trajectory-to-process supervision framework for training agentic vision-language game agents from gameplay trajectories. The central idea is to preserve the temporal structure of gameplay trajectories by representing them as plan-conditioned segments, and to use these segments to construct process-level supervision for planning, acting, and verification. During online evaluation, the trained agent maintains an active plan, generates actions conditioned on both the plan and the current observation, and predicts the execution status to decide whether the plan should continue or be replaced. Figure 1 illustrates the overall pipeline.

3.1 TRAJECTORY SEGMENTATION AND LABEL DEFINITIONS

We first convert each gameplay trajectory into a sequence of local segments and assign supervision labels to each segment. Each segment is associated with a segment-level plan, and each step within the segment is paired with a verification label that describes the execution status of the active plan. The segmentation is based on environment-specific task structure: a segment boundary is placed when the current local objective has been completed or when the trajectory naturally transitions to a new objective. These segments serve as the basic units for constructing planning, action-generation, and verification supervision. Consider a gameplay trajectory

$$\tau = (o_1, a_1, o_2, a_2, \ldots, o_T, a_T, o_{T+1}), \tag{1}$$

where $o_t \in \mathcal{O}$ is a visual observation and $a_t \in \mathcal{A}$ is the action taken at time step t. Instead of treating τ as a collection of independent observation-action pairs, we organize it into an ordered sequence of segments $\tau = (\gamma_1, \gamma_2, \ldots, \gamma_K)$. Each segment γ_k spans a contiguous subsequence of the trajectory,

$$\gamma_k = (o_{b_k}, a_{b_k}, o_{b_k+1}, \ldots, o_{e_k}, a_{e_k}, o_{e_k+1}), \tag{2}$$

and is associated with a segment-level plan p_k. The plan specifies the local objective that governs the actions within the segment. In our instantiation, this plan is derived from the terminal state of the segment, but the formulation only requires that each segment be associated with a plan label.

Algorithm 1 Plan-act-verify Online Decision Loop

Require: initial observation o_1, trained planner Plan, trained actor-verifier Act

$t \leftarrow 1$
$o_{\text{plan}} \leftarrow o_1$
$p \leftarrow \mathrm{Plan}(o_{\text{plan}})$
while the episode has not terminated **do**
  $[s_t, a_t, s_{t+1}, v_t] \leftarrow \mathrm{Act}(o_{\text{plan}}, p, o_t)$
  if $v_t = \texttt{incompatible}$ **then**
    $o_{\text{plan}} \leftarrow o_t$
    $p \leftarrow \mathrm{Plan}(o_{\text{plan}})$
    continue
  end if
  Execute a_t and receive the next observation o_{t+1}
  if $v_t \in \{\texttt{succeeded}, \texttt{failed}\}$ **then**
    $o_{\text{plan}} \leftarrow o_{t+1}$
    $p \leftarrow \mathrm{Plan}(o_{\text{plan}})$
  end if
  $t \leftarrow t + 1$
end while

Within each segment, every step is paired with a verification state that describes the execution status of the active plan. We write the step-wise execution under plan p_k as

$$\mathcal{E}_k = \{(o_t, a_t, v_t)\}_{t=b_k}^{e_k}, \tag{3}$$

where $v_t \in \mathcal{V} = \{\texttt{in-progress}, \texttt{succeeded}, \texttt{failed}, \texttt{incompatible}\}$. The four states form a compact, control-oriented representation of execution progress. `in-progress` indicates that, after the proposed action, the local objective remains achievable but is not yet complete; the current plan is retained. `succeeded` indicates that the proposed action completes the objective, after which the agent replans from the next observation. `failed` indicates that the proposed action makes the objective no longer achievable as intended; the current control loop executes the action and replans from the resulting observation. In contrast, `incompatible` indicates that the active plan is already inconsistent with the current observation before execution; the agent discards the generated action and replans from the current observation.

3.2 Model Inputs, Outputs, and Online Inference

Given the segmented trajectories above, we train a single vision-language model with two prompt formats: a planner prompt and an actor-verifier prompt. The planner generates the active plan from a planning observation, while the actor-verifier predicts the action and execution status conditioned on the active plan and the current observation.

At the beginning of a segment, the planner takes the segment-start observation o_i as input and generates a plan

$$p_i = \mathrm{Plan}(o_i). \tag{4}$$

The generated plan remains active across subsequent steps unless the predicted verification state indicates that replanning is needed.

Given the planning observation o_i, the active plan p_i, and the current observation o_j with $i \leq j$, the actor-verifier predicts an execution tuple

$$[s_j, a_j, s_{j+1}, v_j] = \mathrm{Act}(o_i, p_i, o_j). \tag{5}$$

Here, s_j is a textual description of the current state, a_j is the action to execute at the current step, s_{j+1} is the predicted next-state description after executing the action, and v_j is the predicted status of the active plan.

Algorithm 1 summarizes how the predicted execution status controls plan persistence and replanning during online interaction.

3.3 Basic Training Sample Construction

We next describe how segmented trajectories are converted into basic supervised training samples for planner learning and actor-verifier learning. Let $\mathcal{G}$ denote the set of segments extracted from gameplay trajectories, and let $\mathcal{G}_{\text{opt}} \subset \mathcal{G}$ denote the subset of optimal-plan segments. Each segment provides supervision for planning, plan-conditioned action generation, and verification.

For an optimal-plan segment $\gamma \in \mathcal{G}_{\text{opt}}$ spanning action steps i through n, we derive its plan label p^γ from the terminal observation o_{n+1} and construct planner supervision:

$$o_j \to p^\gamma, \qquad j \in [i, n]. \tag{6}$$

Assigning the same plan label to multiple observations within a segment exposes the planner to different intermediate states that remain consistent with the same local objective.

For action and verification, any segment $\gamma \in \mathcal{G}$ spanning action steps i through n provides step-wise supervision:

$$(o_i, p^\gamma, o_j) \to (s_j, a_j, s_{j+1}, v_j), \qquad j \in [i, n]. \tag{7}$$

Here, the input contains the planning observation o_i, the active plan p^γ, and the current observation o_j. The output contains the current state description, the action, the predicted next-state description, and the verification state.

Each trajectory segment provides two basic verification labels: non-terminal steps are labeled $v_j = \texttt{in-progress}$ for $j \in [i, n-1]$, while the terminal step is labeled $v_n = \texttt{succeeded}$. These samples provide the basic supervision for executing a fixed plan over multiple steps and recognizing when the segment should terminate successfully.

3.4 Augmented Training Sample Construction

The basic samples above cover plan generation, normal execution, and successful segment termination. However, successful trajectories alone provide limited supervision for how actions should change under different plans, how failed execution should be recognized, and how stale plans should be handled. We further construct augmented training samples.

Alternative-plan supervision. To make action generation depend explicitly on the active plan, we construct alternative-plan samples. For a step j in an optimal segment, we replace the original plan p^γ with an alternative feasible plan $\tilde{p}_j^\gamma$ and derive the corresponding execution tuple:

$$(o_i, \tilde{p}_j^\gamma, o_j) \to (s_j, \tilde{a}_j^\gamma, \tilde{s}_{j+1}^\gamma, \tilde{v}_j^\gamma), \tag{8}$$

where $\tilde{v}_j^\gamma = \texttt{in-progress}$. These samples expose the model to cases where the visual observation is unchanged but the active plan differs, requiring the action to change accordingly.

Alternative-action supervision. To provide supervision for failed execution, we construct samples by replacing the original action with an alternative action $\bar{a}_j$ under the same plan:

$$(o_i, p^\gamma, o_j) \to (s_j, \bar{a}_j, \bar{s}_{j+1}, \bar{v}_j). \tag{9}$$

We assign $\bar{v}_j = \texttt{failed}$ when the alternative action makes the current plan no longer achievable as intended, or drives the trajectory into an unrecoverable state without replanning. If the alternative action is still recoverable under the active plan, we assign $\bar{v}_j = \texttt{in-progress}$. These samples provide explicit supervision for the execution-status label `failed`, which is not available from successful segments alone.

Incompatible-plan supervision. To provide supervision for stale or mismatched plans, we pair a current observation with a plan inherited from a previous segment:

$$(o_q, p^{\gamma^-}, o_i) \to (s_i, a_i, s_{i+1}, v_i^{\text{inc}}), \tag{10}$$

where $v_i^{\text{inc}} = \texttt{incompatible}$. Here, γ^- denotes an earlier segment, p^{γ^-} is its associated plan, and o_q is the planning observation from which that stale plan was generated. These samples represent cases in which the active plan no longer matches the current observation. They provide supervision for recognizing that execution should not continue under the outdated plan and that replanning should be triggered from the current state.

3.5 Training Objective and Loss Masking

The constructed data define two learning problems: planning in Eq. 4, and plan-conditioned action generation with verification in Eq. 5. We train a single vision-language model to handle both problems using different prompts and a shared parameterization.

Training proceeds in two stages. In the first stage, the model is trained only on planner samples:

$$\mathcal{L}_{\text{plan}} = - \sum_{(o,p)\in\mathcal{D}_{\text{plan}}} \log P_\theta(p \mid o). \tag{11}$$

This stage initializes the model for segment-level plan generation.

In the second stage, planner samples are retained while action-verification samples are introduced. Let $\mathcal{D}_{\text{av}}$ denote the union of standard action-verification samples and augmented process-supervision samples. Following prior work on selective loss masking for structured supervision (Han et al., 2026; Altinisik et al., 2026), we use a token-level mask for action-verification data. For an input-output pair $(x, y) \in \mathcal{D}_{\text{av}}$, the objective is:

$$\mathcal{L}_{\text{av}} = - \sum_{(x,y)\in\mathcal{D}_{\text{av}}} \sum_{\ell} M_\ell(y) \log P_\theta(y_\ell \mid x, y_{<\ell}), \tag{12}$$

where $M_\ell(y)$ is a token-level loss mask. The mask prevents non-target actions in augmented samples from directly supervising the policy action output. For standard execution samples and alternative-plan samples, all output tokens are supervised. For alternative-action samples and incompatible-plan samples, the loss on action tokens is masked, while the losses on state descriptions and verification states remain active. This allows the model to learn failure and incompatibility signals from augmented data without treating the constructed or mismatched actions as desired policy actions.

4 Game-Specific Instantiations

Our framework requires segment boundaries, plan labels, state descriptions, and verification labels. We instantiate these quantities in each game according to its task structure, using symbolic game states only for automatic data construction. During training and evaluation, the policy receives visual observations and textual prompts as inputs. Additional details are provided in Appendix A.

Tetris. Each segment covers one falling piece from spawn to landing, with a plan specifying its target landing column and rotation.

Sokoban. Trajectories are divided into movement segments between box-pushing events and segments of consecutive pushes. Plans specify a movement destination or a pushing direction and step count.

MiniGrid. Each segment includes movement toward a task-relevant action and its execution. Plans specify the destination, facing direction, and action, such as picking up a key, opening a door, or reaching the goal.

These instantiations use concise, fixed-format local plans generated automatically from the trajectories.

5 Experiments

We evaluate whether trajectory-to-process supervision improves vision-language game agents and whether the learned plan-act-verify process behaves as intended.

5.1 Experimental Setup

We use Qwen3-VL-8B (Bai et al., 2025) as the primary backbone family, initialized from the public Qwen3-VL-8B-Instruct checkpoint. Unless otherwise stated, all compared methods use the same training trajectories, train/test split, backbone, and inference setting. Full details on data statistics, training hyperparameters, inference settings, and prompt templates are provided in Appendix A.

Table 1: Game performance and plan following on Qwen3-VL-8B. For game performance, Tetris reports average cleared lines, while Sokoban and MiniGrid report success rate (%). Plan following is reported as accuracy (%). Higher is better for all metrics. Direct Action has no explicit plan and is therefore not evaluated for plan following.

Method	Game Performance			Plan Following		
	Tetris	Sokoban	MiniGrid	Tetris	Sokoban	MiniGrid
Direct Action	17.81	64	76	–	–	–
Plan-as-CoT	47.98	54	80	96.76	63.66	30.62
Plan-Chunk	16.19	20	91	92.16	29.67	45.45
Ours	**62.42**	**86**	**98**	**99.45**	**98.37**	**100.00**

We compare against three representative training paradigms for vision-language game agents. **Direct Action** predicts the next action directly from the current visual input, following the standard direct action-prediction paradigm used in models such as OpenVLA (Kim et al., 2025), VPT (Baker et al., 2022), and STEVE-1 (Lifshitz et al., 2023). **Plan-as-CoT** generates an intermediate plan before each action step and replans at every step, representing reasoning-before-action formulations such as CoT-VLA (Zhao et al., 2025), ACoT-VLA (Zhong et al., 2026), and Lumine (Tan et al., 2025a). **Plan-Chunk** first predicts a high-level plan and then executes an action chunk before replanning, following the style adopted in methods such as Optimus-2 (Li et al., 2025b) and OpenHA (Wang et al., 2025a).

5.2 Game Performance

We first evaluate game performance across the three environments. For task performance, Tetris is reported by the average number of cleared lines over 100 episodes, while Sokoban and MiniGrid are reported by success rate over 50 and 100 test maps, respectively. Table 1 reports the results on Qwen3-VL-8B, including both task performance and plan following. Our method achieves the best task performance on all three environments, reaching 62.42 cleared lines in Tetris, 86% success on Sokoban, and 98% success on MiniGrid. Compared with Direct Action, the gains indicate that trajectory-derived process supervision provides useful information beyond local action imitation. Compared with Plan-as-CoT and Plan-Chunk, the results suggest that plans are more effective when their execution and verification are explicitly supervised over time.

The comparison with Plan-Chunk further highlights the importance of closed-loop execution. While chunked execution remains competitive on MiniGrid, it performs substantially worse on Tetris and Sokoban, where early errors in an open-loop action sequence can affect later steps. By re-evaluating the current observation under the active plan at every step, our method keeps action generation conditioned on the latest game state.

5.3 Evaluating Plan Following

To better understand how different formulations use plans during execution, we evaluate plan following under environment-specific criteria. For our method and Plan-Chunk, a plan is intended to govern a multi-step segment, so we measure whether the realized terminal outcome matches the plan target. For Plan-as-CoT, the plan is regenerated at every step and functions as a step-wise prefix, so we measure whether the action taken at each step is consistent with the current plan.

As shown in Table 1, our method achieves the strongest plan following on all three environments. This result indicates that explicit supervision of execution under a plan leads to tighter alignment between plans and behavior than step-wise replanning or open-loop chunk execution.

5.4 Plan Transition Prediction

We further evaluate whether the learned verification states can support plan transitions. Table 2 reports the confusion matrix of execution-status prediction on Tetris. The dominant states, `in-progress` and `succeeded`, are predicted almost perfectly, indicating that the model can identify when the current plan should continue and when it has been completed. The rarer transition-related states,

Table 2: Confusion matrix of execution-status prediction for our method on Tetris.

Pred. \ GT	In-progress	Succeeded	Failed	Incompatible
In-progress	**56536**	10	5	1
Succeeded	0	**13994**	1	0
Failed	0	0	**71**	0
Incompatible	1	0	6	**16**

Table 3: Generalization across vision-language backbones. Tetris is reported as average cleared lines, while Sokoban and MiniGrid are reported as success rate (%). Higher is better.

Backbone	Method	Tetris	Sokoban	MiniGrid
InternVL3-8B	Direct Action	17.04	50	85
	Ours	**47.26**	**80**	**100**
Gemma3-12B	Direct Action	3.94	54	48
	Ours	**6.17**	**86**	**99**

Table 4: Ablation studies on Qwen3-VL-8B. Tetris is reported as average cleared lines, while Sokoban and MiniGrid are reported as success rate (%). Higher is better. SP: suboptimal-plan supervision; AP: alternative-plan supervision; AA: alternative-action supervision; IP: incompatible-plan supervision.

Method	Tetris	Sokoban	MiniGrid
Ours	**62.42**	**86**	**98**
w/o selective masking	55.58	32	53
w/o SP & AP supervision	52.90	48	96
w/o AA & IP supervision	53.44	56	97

Table 5: Inference-time replanning interventions on Tetris using the same trained checkpoint. The fixed-period strategy replans every 5 environment steps. Performance is measured by average cleared lines.

Replanning strategy	Cleared lines
Learned verifier-controlled replanning	62.42
Replan at every step	55.56
Replan every 5 steps (fixed period)	5.13

`failed` and `incompatible`, are also recognized with high recall, although a small number of failed cases are confused with `incompatible`. Overall, these results show that the learned verification signal can support closed-loop decisions about continuing, terminating, or replacing the active plan.

5.5 Generalization across Vision-Language Backbones

To test whether the proposed training formulation is specific to one model family, we further evaluate InternVL3-8B and Gemma3-12B. As shown in Table 3, our method consistently improves over Direct Action on both backbones across all three environments. These results suggest that the benefit of trajectory-to-process supervision transfers across different vision-language backbones.

5.6 Ablation Studies

Table 4 reports ablation results on Qwen3-VL-8B. Removing selective loss masking causes clear performance drops, especially on Sokoban and MiniGrid. This indicates that augmented samples are useful only when their non-target actions are prevented from directly entering the policy target. Removing suboptimal-plan and alternative-plan supervision weakens the training signal that ties action selection to the active plan, making it harder for the model to distinguish how execution should

Table 6: Sensitivity to synthetic training-data corruption on Tetris.

Corruption rate	0%	10%	30%	50%
Cleared lines	62.42	66.61	65.95	54.66

Table 7: Sensitivity to trajectory segmentation on MiniGrid.

Segmentation	Success rate (%)
Original task-structured	98
Fixed-length	100
Random	100

vary under different plans. Removing alternative-action and incompatible-plan supervision weakens the model's ability to recognize failed execution and stale plans, both of which are important for triggering replanning.

To isolate the inference-time contribution of the predicted status, we keep the trained checkpoint fixed and compare three replanning strategies on Tetris: the learned verifier-controlled policy, replanning at every environment step, and fixed-period replanning every 5 environment steps. In the two alternative strategies, replanning is scheduled by the prescribed interval rather than by the predicted verification state. Table 5 reports the results. Using the learned status to control plan transitions obtains 62.42 cleared lines, compared with 55.56 when replanning at every step and 5.13 when replanning every 5 steps. This intervention evaluates the operational value of using the predicted status to control plan persistence.

5.7 Sensitivity to Process Annotations

Annotation-noise tolerance. To examine sensitivity to imperfect process annotations, we replace 10%, 30%, and 50% of the Tetris training data with samples containing random plans/actions. As shown in Table 6, the resulting scores are 66.61, 65.95, and 54.66 cleared lines, compared with 62.42 using the original data. Under the tested corruption procedure, the scores remain comparable at 10–30% corruption and decrease at 50%. These results support tolerance to the annotation noise.

Segmentation sensitivity. We also evaluate alternative trajectory segmentations on MiniGrid. In addition to the original task-structured segmentation, we train variants using fixed-length and random segmentation. Both variants achieve 100% success, compared with 98% for the original segmentation (Table 7). These results indicate that the formulation does not depend on one exact hand-designed boundary rule in the evaluated MiniGrid setting.

6 Conclusion

We introduced a trajectory-to-process supervision framework that jointly trains planning, plan-conditioned action generation, and execution-status prediction within a single vision-language policy. Experiments on Tetris, Sokoban, and MiniGrid show improved task performance and plan following over direct action prediction, plan-as-CoT, and plan-chunk baselines. Ablations further support the roles of plan-conditioned supervision, verification-controlled replanning, and selective loss masking in learning closed-loop gameplay.

Limitations and Future Work. Our current data construction relies on environment-specific solvers, symbolic states, and hand-designed rules. Although the framework can accommodate other sources of process annotations, obtaining reliable labels without these resources remains a challenge. We have not established generalization to human gameplay demonstrations, modern open-ended games, or real-world visual-agent tasks. Richer reasoning-based plans and explicit reflection also remain unevaluated. Future work will explore weaker annotation sources and mechanisms that generate corrective feedback to revise deficient plans or actions.

AI USE STATEMENT

We used generative AI tools to assist with implementing the methods, exploring relevant literature, organizing the manuscript, and editing the English text. The authors reviewed and revised AI-assisted code and manuscript content and take full responsibility for the implementation, reported results, and final manuscript.

REFERENCES

Enes Altinisik, Masoomali Fatehkia, Fatih Deniz, Nadir Durrani, Majd Hawasly, Mohammad Raza, and Husrev Taha Sencar. Do I Really Know? learning factual self-verification for hallucination reduction. *arXiv preprint arXiv:2602.02018*, 2026.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, et al. Qwen3-VL technical report. *arXiv preprint arXiv:2511.21631*, 2025.

Bowen Baker, Ilge Akkaya, Peter Zhokov, Joost Huizinga, Jie Tang, Adrien Ecoffet, Brandon Houghton, Raul Sampedro, and Jeff Clune. Video pretraining (VPT): Learning to act by watching unlabeled online videos. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24639–24654, 2022.

Peng Chen, Pi Bu, Yingyao Wang, Xinyi Wang, Ziming Wang, Jie Guo, Yingxiu Zhao, Qi Zhu, Jun Song, Siran Yang, Jiamang Wang, and Bo Zheng. CombatVLA: An efficient vision-language-action model for combat tasks in 3d action role-playing games. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 10919–10928, October 2025.

Tingjun Dai, Mingfei Han, Tingwen Du, Zhiheng Liu, Zhihui Li, Salman Khan, Jun Yu, and Xiaojun Chang. See, plan, rewind: Progress-aware vision-language-action models for robust robotic manipulation. *arXiv preprint arXiv:2603.09292*, 2026.

Yiming Ding, Carlos Florensa, Mariano Phielipp, and Pieter Abbeel. Goal-conditioned imitation learning. In *Advances in Neural Information Processing Systems*, volume 32, pp. 15298–15309, 2019.

Jiahui Fu, Junyu Nan, Lingfeng Sun, Hongyu Li, Jianing Qian, Jennifer L. Barry, Kris Kitani, and George Konidaris. NovaPlan: Zero-shot long-horizon manipulation via closed-loop video language planning. *arXiv preprint arXiv:2602.20119*, 2026.

Abhishek Gupta, Vikash Kumar, Corey Lynch, Sergey Levine, and Karol Hausman. Relay policy learning: Solving long-horizon tasks via imitation and reinforcement learning. In Leslie Pack Kaelbling, Danica Kragic, and Komei Sugiura (eds.), *Proceedings of the Conference on Robot Learning*, volume 100 of *Proceedings of Machine Learning Research*, pp. 1025–1037. PMLR, 2020.

Jinyi Han, Zixiang Di, Zishang Jiang, Ying Liao, Jiaqing Liang, Yongqi Wang, and Yanghua Xiao. Structured reasoning for large language models. *arXiv preprint arXiv:2601.07180*, 2026.

Brian Ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander T. Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Mengyuan Yan, Noah Brown, Michael Ahn, Omar Cortes, Nicolas Sievers, Clayton Tan, Sichun Xu, Diego Reyes, Jarek Rettinghouse, Jornell Quiambao, Peter Pastor, Linda Luu, Kuang-Huei Lee, Yuheng Kuang, Sally Jesmonth, Nikhil J. Joshi, Kyle Jeffrey, Rosario Jauregui Ruano, Jasmine Hsu, Keerthana Gopalakrishnan, Byron David, Andy Zeng, and Chuyuan Kelly Fu. Do As I Can, Not As I Say: Grounding language in robotic affordances. In Karen Liu, Dana Kulic, and Jeff Ichnowski (eds.), *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 287–318. PMLR, 2023. URL https://proceedings.mlr.press/v205/ichter23a.html.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan P. Foster, Pannag R. Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. OpenVLA: An open-source vision-language-action model. In *Proceedings of The 8th Conference on Robot Learning*, volume 270 of *Proceedings of Machine Learning Research*, pp. 2679–2713. PMLR, 2025.

Jacky Kwok, Xilun Zhang, Mengdi Xu, Yuejiang Liu, Azalia Mirhoseini, Chelsea Finn, and Marco Pavone. Scaling verification can be more effective than scaling policy learning for vision-language-action alignment. *arXiv preprint arXiv:2602.12281*, 2026. doi: 10.48550/arXiv.2602.12281.

Hoang Le, Nan Jiang, Alekh Agarwal, Miroslav Dudik, Yisong Yue, and Hal Daumé, III. Hierarchical imitation and reinforcement learning. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 2917–2926. PMLR, 2018.

Muyao Li, Zihao Wang, Kaichen He, Xiaojian Ma, and Yitao Liang. JARVIS-VLA: Post-training large-scale vision language models to play visual games with keyboards and mouse. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 17878–17899, Vienna, Austria, July 2025a. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.920.

Zaijing Li, Yuquan Xie, Rui Shao, Gongwei Chen, Dongmei Jiang, and Liqiang Nie. Optimus-2: Multimodal minecraft agent with goal-observation-action conditioned policy. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9039–9049, June 2025b.

Shalev Lifshitz, Keiran Paster, Harris Chan, Jimmy Ba, and Sheila McIlraith. STEVE-1: A generative model for text-to-behavior in minecraft. In *Advances in Neural Information Processing Systems*, volume 36, pp. 69900–69929, 2023.

Fanqi Lin, Ruiqian Nai, Yingdong Hu, Jiacheng You, Junming Zhao, and Yang Gao. OneTwoVLA: A unified vision-language-action model with adaptive reasoning. *arXiv preprint arXiv:2505.11917*, 2025.

Yuanzhe Liu, Jingyuan Zhu, Yuchen Mo, Gen Li, Xu Cao, Jin Jin, Yifan Shen, Zhengyuan Li, Tianjiao Yu, Wenzhen Yuan, Fangqiang Ding, and Ismini Lourentzou. PALM: Progress-aware policy learning via affordance reasoning for long-horizon robotic manipulation. *arXiv preprint arXiv:2601.07060*, 2026. doi: 10.48550/arXiv.2601.07060. CVPR 2026.

Corey Lynch, Mohi Khansari, Ted Xiao, Vikash Kumar, Jonathan Tompson, Sergey Levine, and Pierre Sermanet. Learning latent plans from play. In Leslie Pack Kaelbling, Danica Kragic, and Komei Sugiura (eds.), *Proceedings of the Conference on Robot Learning*, volume 100 of *Proceedings of Machine Learning Research*, pp. 1113–1132. PMLR, 2020.

SIMA Team, Adrian Bolton, Alexander Lerchner, Alexandra Cordell, Alexandre Moufarek, Andrew Bolt, Andrew Lampinen, Anna Mitenkova, Arne Olav Hallingstad, Bojan Vujatovic, Bonnie Li, Cong Lu, Daan Wierstra, Daniel P. Sawyer, Daniel Slater, David Reichert, Davide Vercelli, Demis Hassabis, Drew A. Hudson, Duncan Williams, Ed Hirst, Fabio Pardo, Felix Hill, Frederic Besse, Hannah Openshaw, Harris Chan, Hubert Soyer, Jane X. Wang, Jeff Clune, John Agapiou, John Reid, Joseph Marino, Junkyung Kim, Karol Gregor, Kaustubh Sridhar, Kay McKinney, Laura Kampis, Lei M. Zhang, Loic Matthey, Luyu Wang, Maria Abi Raad, Maria Loks-Thompson, Martin Engelcke, Matija Kecman, Matthew Jackson, Maxime Gazeau, Ollie Purkiss, Oscar Knagg, Peter Stys, Piermaria Mendolicchio, Raia Hadsell, Rosemary Ke, Ryan Faulkner, Sarah Chakera, Satinder Singh Baveja, Shane Legg, Sheleem Kashem, Tayfun Terzi, Thomas Keck, Tim Harley, Tim Scholtes, Tyson Roberts, Volodymyr Mnih, Yulan Liu, Zhengdong Wang, and Zoubin Ghahramani. SIMA 2: A generalist embodied agent for virtual worlds. *arXiv preprint arXiv:2512.04797*, 2025.

Weihao Tan, Xiangyang Li, Yunhao Fang, Heyuan Yao, Shi Yan, Hao Luo, Tenglong Ao, Huihui Li, Hongbin Ren, Bairen Yi, Yujia Qin, Bo An, Libin Liu, and Guang Shi. Lumine: An open recipe for building generalist agents in 3d open worlds. *arXiv preprint arXiv:2511.08892*, 2025a.

Weihao Tan, Wentao Zhang, Xinrun Xu, Haochong Xia, Ziluo Ding, Boyu Li, Bohan Zhou, Junpeng Yue, Jiechuan Jiang, Yewen Li, Ruyi An, Molei Qin, Chuqiao Zong, Longtao Zheng, Yujie Wu, Xiaoqiang Chai, Yifei Bi, Tianbao Xie, Pengjie Gu, Xiyun Li, Ceyao Zhang, Long Tian, Chaojie Wang, Xinrun Wang, Börje F. Karlsson, Bo An, Shuicheng Yan, and Zongqing Lu. Cradle: Empowering foundation agents towards general computer control. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 58658–58725. PMLR, 2025b.

Chaoyang Wang, Wenrui Bao, Sicheng Gao, Bingxin Xu, Yu Tian, Yogesh S. Rawat, Yunhao Ge, and Yuzhang Shang. VLA-Thinker: Boosting vision-language-action models through thinking-with-image reasoning. *arXiv preprint arXiv:2603.14523*, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, explain, plan and select: Interactive planning with LLMs enables open-world multi-task agents. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Zihao Wang, Muyao Li, Kaichen He, Xiangyu Wang, Zhancun Mu, Anji Liu, and Yitao Liang. OpenHA: A series of open-source hierarchical agentic models in minecraft. *arXiv preprint arXiv:2509.13347*, 2025a.

Zihao Wang, Xujing Li, Yining Ye, Junjie Fang, Haoming Wang, Longxiang Liu, Shihao Liang, Junting Lu, Zhiyong Wu, Jiazhan Feng, Wanjun Zhong, Zili Li, Yu Wang, Yu Miao, Bo Zhou, Yuanfan Li, Hao Wang, Zhongkai Zhao, Faming Wu, Zhengxuan Jiang, Weihao Tan, Heyuan Yao, Shi Yan, Xiangyang Li, Yitao Liang, Yujia Qin, and Guang Shi. Game-TARS: Pretrained foundation models for scalable generalist multimodal game agents. *arXiv preprint arXiv:2510.23691*, 2025b.

Zhejian Yang, Yongchao Chen, Xueyang Zhou, Jiangyue Yan, Dingjie Song, Yinuo Liu, Yuting Li, Yu Zhang, Pan Zhou, Hechang Chen, and Lichao Sun. Agentic robot: A brain-inspired framework for vision-language-action models in embodied agents. *arXiv preprint arXiv:2505.23450*, 2025.

Kuan Zhang, Dongchen Liu, Qiyue Zhao, Jinkun Hou, Xinran Zhang, Qinlei Xie, Miao Liu, and Yiming Li. Gameverse: Can vision-language models learn from video-based reflection? *arXiv preprint arXiv:2603.06656*, 2026.

Qingqing Zhao, Yao Lu, Moo Jin Kim, Zipeng Fu, Zhuoyang Zhang, Yecheng Wu, Zhaoshuo Li, Qianli Ma, Song Han, Chelsea Finn, Ankur Handa, Tsung-Yi Lin, Gordon Wetzstein, Ming-Yu Liu, and Donglai Xiang. CoT-VLA: Visual chain-of-thought reasoning for vision-language-action models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1702–1713, June 2025.

Linqing Zhong, Yi Liu, Yifei Wei, Ziyu Xiong, Maoqing Yao, Si Liu, and Guanghui Ren. ACoT-VLA: Action chain-of-thought for vision-language-action models. *arXiv preprint arXiv:2601.11404*, 2026. doi: 10.48550/arXiv.2601.11404. Accepted by CVPR 2026.

A Experimental Details

This appendix provides additional details on the data collection procedure, training configuration, inference protocol, and prompt templates used in our experiments, as well as an online rollout example.

A.1 Environment and Data Collection

We collect trajectories separately for Tetris, Sokoban, and MiniGrid. In all environments, trajectories are generated by environment-specific solvers and are used to construct planner samples, action-verification samples, and augmented supervision. We use disjoint random seeds for training and evaluation so that the generated game instances do not overlap. Each trajectory contains both visual

observations and symbolic game states. Symbolic states are used only for automatic data construction and label generation, including segment boundaries, plan labels, state descriptions, alternative-plan action labels, and verification labels. During model training and evaluation, the policy receives visual observations and textual prompts as inputs.

These environment-specific resources provide a low-cost offline annotation route rather than an architectural requirement of the plan-act-verify policy. The training formulation can also accommodate human gameplay trajectories with suitable process annotations, although obtaining such annotations may require manual effort when symbolic information is unavailable. Our contribution is a visual agent training formulation rather than a general solution to process annotation.

The plan labels used here are concise, fixed-format local subtask specifications, rather than reasoning-rich plans. This format reflects our low-cost annotation strategy, not a restriction of the model interface. When richer annotations are available, a plan may additionally contain a human-annotated or LLM-expanded reasoning prefix before the executable objective, provided that the same representation is supplied to the actor-verifier during execution. The current experiments do not evaluate this extension.

Table 8 summarizes the trajectory statistics. Figure 2 shows representative visual observations from the three environments.

Table 8: Data collection statistics for the three environments.

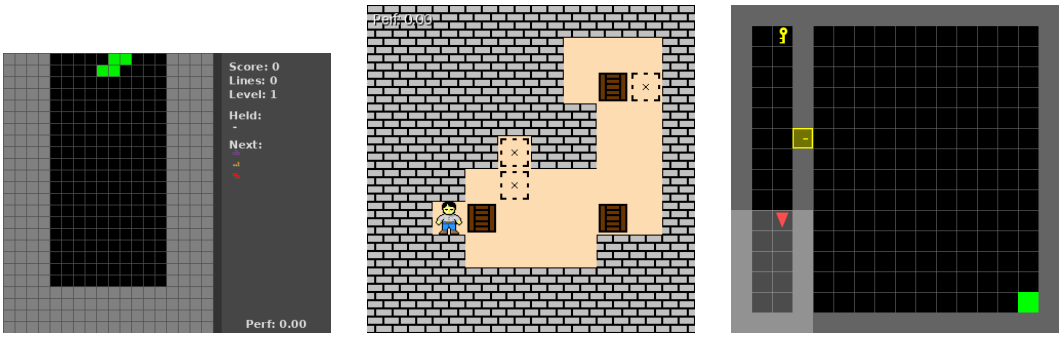
Environment	Environment source	Expert solver	Number of trajectories
Tetris	GamingAgent	Heuristic solver	1774
Sokoban	gym_sokoban	BFS solver	1121
MiniGrid	Gymnasium MiniGrid	Rule-based solver	288

For Sokoban and MiniGrid, movement-related plans are verbalized using coordinates relative to the current player or agent position. When planner supervision is constructed from intermediate observations within the same segment, the underlying segment objective is shared, but its textual coordinate representation is recomputed with respect to the observation used as the planner input. Thus, assigning a segment-level plan to multiple observations should be understood as assigning the same local objective, not necessarily reusing the identical relative-coordinate string across all observations.

For Tetris, we use the game environment implemented in the GamingAgent repository.[1] We collect 1774 trajectories using a heuristic solver. Each trajectory starts from a randomly sampled initial state and has a maximum horizon of 100 steps. The 100-step horizon is used only for collecting training trajectories, while evaluation runs for up to 1000 environment steps to measure sustained gameplay. We define each segment as the full execution of one falling piece, from its spawn state to its final landing state. The segment-level plan specifies the target landing column and rotation, while the state description records the current piece position and orientation. For alternative-plan augmentation, we randomly sample a legal landing placement as the alternative plan and derive the corresponding action using the environment transition rules. For alternative-action augmentation, we label the verification state as `failed` when the alternative action is `hard_drop`, since it immediately commits the piece to a landing position and can make the active placement plan unreachable.

For Sokoban, we use the `gym_sokoban` package to generate random 10×10 maps. We solve the generated maps using breadth-first search (BFS) and collect 1121 expert trajectories. We decompose trajectories into movement segments between box-pushing events and pushing segments that contain consecutive push actions. The segment-level plan specifies either a movement destination or a box-pushing direction together with the required number of pushing steps, while the state description records the player position and relevant box configuration. For alternative-plan augmentation, we randomly sample a legal movement target or a legal box-pushing direction with a step count, and compute the corresponding action using shortest-path search or deterministic transition rules. For alternative-action augmentation, we label the verification state as `failed` when an alternative action during a movement plan causes a box to move, since it changes the box configuration and invalidates the active movement plan.

[1] `https://github.com/lmgame-org/GamingAgent`



(a) Tetris **(b) Sokoban** **(c) MiniGrid**

Figure 2: Representative visual observations from the three game environments used in our experiments. In Tetris, the agent places falling tetrominoes to clear rows. In Sokoban, the agent pushes boxes onto target locations while avoiding deadlocks. In MiniGrid, the agent navigates a grid world with task-relevant objects such as keys, doors, and goals. These rendered observations are provided to the vision-language policy as visual inputs during training and evaluation.

For MiniGrid, we use the `gymnasium` implementation of `MiniGrid-DoorKey-16x16-v0`. We generate random maps and solve them using a rule-based expert policy, resulting in 288 expert trajectories. We define each segment as the movement leading to a discrete task action and the execution of that action. The segment-level plan specifies the movement destination, facing direction, and task action, such as picking up a key, toggling a door, or reaching the goal. The state description records the agent position and orientation. For alternative-plan augmentation, we randomly sample a legal movement destination, facing direction, and task action when applicable, and derive the corresponding action using the rule-based transition structure. We do not construct `failed` alternative-action samples for MiniGrid, because the considered actions do not create irreversible plan-breaking states in this environment.

Across all environments, we obtain suboptimal-plan supervision by occasionally replacing the solver's optimal plan with a randomly sampled feasible plan and then rolling out a valid trajectory under that plan. These trajectories provide supervision for execution under weaker but feasible planning decisions. Incompatible-plan samples are constructed by pairing the current observation with a plan from a previous segment whose objective is no longer consistent with the current state. These samples are labeled as `incompatible` and are used to supervise pre-action replanning.

For Tetris, plan following is correct if the final landing column and rotation match the generated plan. For Sokoban, it is correct if the agent reaches the planned movement target or completes the planned push direction and step count. For MiniGrid, it is correct if the agent reaches the planned relative destination with the planned orientation and executes the intended task action.

Training samples and matched-data control. The original Tetris baselines use 100,000 image-action samples obtained from 1,774 trajectories. Our method uses 100,000 planner samples and 40,392 actor-verifier samples. Although augmentation can produce additional actor-verifier examples, 40,392 is the number actually used in the reported training run.

To examine whether the improvement can be explained by sample count or planner multi-task training alone, we construct a matched multi-task baseline. It retains the same 100,000 planner samples and replaces the 40,392 actor-verifier samples with 40,392 ordinary image-action samples. Table 9 reports the resulting performance.

The matched multi-task baseline improves over the original Direct Action baseline but remains below our method. Thus, matching the numbers of planner and execution examples does not remove the performance gap, supporting the value of plan-conditioned execution and verification supervision beyond planner multi-task training alone.

Table 9: Tetris performance and numbers of training examples. Planner and execution columns count examples in the respective training sets, not cumulative presentations across epochs. The execution examples are ordinary image-action pairs for the first two rows and actor-verifier examples for our method.

Method	Planner	Execution	Cleared lines
Direct Action	0	100,000	17.81
Planner + ordinary action	100,000	40,392	24.28
Ours	100,000	40,392	62.42

A.2 Existing Assets and Licenses

We use publicly available environments, software packages, training frameworks, and pretrained vision-language model weights. Table 10 summarizes the existing assets used in our experiments, their usage, and their stated licenses or terms of use. We use these assets according to their respective licenses and terms. We do not redistribute the original environment packages, training framework, or pretrained model weights with this submission.

Table 10: Existing assets used in our experiments.

Asset	Usage in this work	Creator / owner	License or terms
GamingAgent / LMGame Bench	Tetris environment implementation	lmgame-org	MIT License
gym_sokoban	Sokoban environment generation	Max-Philipp Schrader	MIT License
Gymnasium	Environment API and interface	Farama Foundation / OpenAI	MIT License
MiniGrid	MiniGrid-DoorKey environment	Farama Foundation	Apache License 2.0
LLaMA-Factory	Full fine-tuning framework	hiyouga / LLaMA-Factory contributors	Apache-2.0 License
Qwen3-VL-8B-Instruct	Pretrained VLM backbone	Qwen / Alibaba Cloud	Apache-2.0 License
InternVL3-8B	Pretrained VLM backbone	OpenGVLab	MIT License; Qwen2.5 component under Apache-2.0
Gemma3-12B-IT	Pretrained VLM backbone	Google	Gemma Terms of Use

A.3 Training Configuration

All training stages are performed with full-parameter SFT using LLaMA-Factory[2]. Before task-specific training, all backbones first undergo a spatial-ability SFT warm-up stage for 1 epoch using synthetic coordinate-QA data. These coordinate-QA tasks are constructed to strengthen the model's understanding of spatial positions and relative coordinates in game images, which is important for interpreting board states, object locations, and movement targets in the evaluated environments.

After the spatial-ability warm-up, we perform task-specific fine-tuning for each method. For our method, training follows the two-stage curriculum described in the main text: the first stage trains the model on planner samples for 3 epochs, and the second stage jointly trains on planner and action-verification samples for 1 epoch. All baseline methods are trained for 3 epochs using their corresponding training data. Unless otherwise stated, all stages use the same optimizer and hyperparameters, as summarized in Table 11.

We use AdamW with a learning rate of 1×10^{-5}, cosine learning-rate scheduling, a warmup ratio of 0.1, and a batch size of 128. Training is performed on 8 GPUs and can be completed within several hours for each training run.

A.4 Inference Configuration

During evaluation, we use the default `generation_config` provided by each pretrained backbone for text generation. The model output is parsed according to the required response format in each prompt. For direct-action baselines, we extract the predicted action from the generated response.

[2] `https://github.com/hiyouga/LLaMA-Factory`

Table 11: Training hyperparameters.

Hyperparameter	Value
Training framework	LLaMA-Factory
Fine-tuning method	Full-parameter SFT
Optimizer	AdamW
Learning rate	1×10^{-5}
Spatial warm-up epochs	1
Task-specific epochs	Ours: 3 planner + 1 joint; baselines: 3
Learning-rate scheduler	Cosine
Warmup ratio	0.1
Batch size	128
Number of GPUs	8
Training time	Several hours per training run

For plan-as-CoT baselines, we extract both the generated plan and the next action. For plan-chunk baselines, we extract the generated plan and the predicted action sequence. For our method, we extract the current state description, action, predicted future state description, and plan status.

For our method, when the predicted plan status is `in progress`, the active plan is retained and reused at the next step. When the predicted status is `succeeded` or `failed`, a new plan is generated from the next observation. When the predicted status is `incompatible`, corresponding to the `incompatible` state in the main paper, the current plan is discarded and a new plan is generated from the current observation before executing further actions.

A.5 Prompt Templates

We provide the prompt templates used for our method and all baselines below. The templates are grouped by environment and method. The following templates use concise structured plans because these labels can be generated automatically and consistently in the three evaluated environments.

A.5.1 Tetris Prompts

Planner prompt.

```
<image>Analyze this Tetris game state and plan the optimal placement. Tetris Game Mechanics:

1. The game board is a grid where pieces (tetrominoes) fall from the top.
2. Your goal is to clear lines by filling all cells in a row, which removes that row.
3. The game ends if the pieces stack up to the top of the board.

Here is the recommended policy to play the game: Keep the board flat and avoid tall stacks.
Prefer filling holes and completing lines. Rotate only when it improves fit.
You should answer with the following format:

Optimal Placement: column [number], row [number], [keep current orientation/rotate left/rotate
      right/rotate 180 degrees]

The block coordinates is defined as follows:

start from bottom left, start from 1, column defined by leftest unit of the block, row defined
      by bottom unit of the block.
```

Actor-verifier prompt.

```
<image>Analyze this Tetris game state and output the optimal next move according to the
     planner generated optimal placement plan, then verify the plan status. Tetris Game
     Mechanics:

1. The game board is a grid where pieces (tetrominoes) fall from the top.
2. Your goal is to clear lines by filling all cells in a row, which removes that row.
3. The game ends if the pieces stack up to the top of the board.

Available Actions (Format: move: (action_name_from_list, frame_count) where frame_count is
     1-10 and action_name_from_list MUST be one of the following exact strings):
```

```
- "no_op": Do nothing for this step.
- "left": Move the current piece one cell to the left.
- "right": Move the current piece one cell to the right.
- "rotate_left": Rotate the current piece 90 degrees counter-clockwise.
- "rotate_right": Rotate the current piece 90 degrees clockwise.
- "soft_drop": Move the current piece one cell down.
- "hard_drop": Instantly drop the current piece to its lowest possible position.

Here is the planner observation and its plan:
<image>

Optimal Placement: column [number], row [number], [keep current orientation/rotate left/rotate
      right/rotate 180 degrees]

The block coordinates is defined as follows:

start from bottom left, start from 1, column defined by leftest unit of the block, row defined
      by bottom unit of the block.

You should first describe the current position and orientation of the falling block and output
      an action to move towards the planned optimal placement.
Then describe the future position and orientation of the falling block at the next game state.
Finally you need to analyze whether the plan will be in progress, succeeded, failed or out of
     date after this action happens.
You should answer with the following format:
Current state: column [number], row [number], orientation [0/1/2/3]
move: (action_name_from_list, frame_count)
Future state: column [number], row [number], orientation [0/1/2/3]
Plan [in progress/succeeded/failed/out of date]
```

Direct-action baseline prompt.

```
<image>Analyze this Tetris game state and plan the optimal next move. Tetris Game Mechanics:

1. The game board is a grid where pieces (tetrominoes) fall from the top.
2. Your goal is to clear lines by filling all cells in a row, which removes that row.
3. The game ends if the pieces stack up to the top of the board.

Available Actions (Format: move: (action_name_from_list, frame_count) where frame_count is
     1-10 and action_name_from_list MUST be one of the following exact strings):
- "no_op": Do nothing for this step.
- "left": Move the current piece one cell to the left.
- "right": Move the current piece one cell to the right.
- "rotate_left": Rotate the current piece 90 degrees counter-clockwise.
- "rotate_right": Rotate the current piece 90 degrees clockwise.
- "soft_drop": Move the current piece one cell down.
- "hard_drop": Instantly drop the current piece to its lowest possible position.

Here is the recommended policy to play the game: Keep the board flat and avoid tall stacks.
     Prefer filling holes and completing lines. Rotate only when it improves fit. Move pieces
     horizontally to good landing positions. Avoid unnecessary rotations or staying in the
     center

Format your response exactly as follows:
move: (action_name_from_list, frame_count)
```

Plan-as-CoT prompt.

```
<image>Analyze this Tetris game state, plan the optimal placement, and output the optimal next
      move. Tetris Game Mechanics:

1. The game board is a grid where pieces (tetrominoes) fall from the top.
2. Your goal is to clear lines by filling all cells in a row, which removes that row.
4. The game ends if the pieces stack up to the top of the board.

Available Actions (Format: move: (action_name_from_list, frame_count) where frame_count is
     1-10 and action_name_from_list MUST be one of the following exact strings):
- "no_op": Do nothing for this step.
- "left": Move the current piece one cell to the left.
- "right": Move the current piece one cell to the right.
- "rotate_left": Rotate the current piece 90 degrees counter-clockwise.
- "rotate_right": Rotate the current piece 90 degrees clockwise.
- "soft_drop": Move the current piece one cell down.
- "hard_drop": Instantly drop the current piece to its lowest possible position.

You should answer with the following format:
Optimal Placement: column [number], row [number], [keep current orientation/rotate left/rotate
      right/rotate 180 degrees]
```

```
move: (action_name_from_list, frame_count)

The block coordinates is defined as follows:
start from bottom left, start from 1, column defined by leftest unit of the block, row defined
      by bottom unit of the block.
```

Plan-Chunk prompt.

```
Analyze this Tetris game state, plan the optimal placement, and output the optimal next move.
     Tetris Game Mechanics:

1. The game board is a grid where pieces (tetrominoes) fall from the top.
2. Your goal is to clear lines by filling all cells in a row, which removes that row.
4. The game ends if the pieces stack up to the top of the board.

Available Actions (Format: move: action_name_from_list where action_name_from_list MUST be one
      of the following exact strings):
- "no_op": Do nothing for this step.
- "left": Move the current piece one cell to the left.
- "right": Move the current piece one cell to the right.
- "rotate_left": Rotate the current piece 90 degrees counter-clockwise.
- "rotate_right": Rotate the current piece 90 degrees clockwise.
- "soft_drop": Move the current piece one cell down.
- "hard_drop": Instantly drop the current piece to its lowest possible position.

You should answer with the following format:
Optimal Placement: column [number], row [number], [keep current orientation/rotate left/rotate
      right/rotate 180 degrees]
move: (action_name_from_list, action_name_from_list, ...)

The block coordinates is defined as follows:
start from bottom left, start from 1, column defined by leftest unit of the block, row defined
      by bottom unit of the block.

Example answer:
Optimal Placement: column 1, row 1, rotate right
move: (rotate_right, left, left, hard_drop)

Here is the current observation:<image>
```

A.5.2 SOKOBAN PROMPTS

Planner prompt.

```
<image>
You are an intelligent AI player solving a Sokoban puzzle.
Your goal is to make strategic moves to push all boxes onto their designated target locations.
Sokoban Game Quick Guide:

Primary Goal: Push all brown boxes onto all target locations. A box on a target becomes yellow.


Game Mechanics & Action Choice:
- The game is played on a grid using the following characters:
- To push a Box: If the Player is adjacent to a Box AND you want to move that Box, you can use
      actions:
 "up", "down", "left", "right" against the box to push it.
  The square beyond the box must be an empty Floor or empty Target.
- Player cannot push a Box into a Wall or another Box.
- The game is solved when all boxes are on target locations (all boxes are yellow).

Key Strategies:
1. Box pushing order: Think carefully about which boxes to push first.
2. Avoid deadlocks: Do not push boxes into corners (unless it is a target) or against walls
     where they cannot be moved out.
3. Player positioning: Position the player to push boxes effectively.

Your response format MUST be the following two types:

Move to the location [Down/Up] [number], [Left/Right] [number] relative to the role.
or
Push [down/up/left/right] for [number] steps.

Example answer:
Move to the location [Down/Up] [number], [Left/Right] [number] relative to the role.
```

Actor-verifier prompt.

```
<image>
You are an intelligent AI player solving a Sokoban puzzle.
Your goal is to make strategic moves to push all boxes onto their designated target locations.
Sokoban Game Quick Guide:

Primary Goal: Push all brown boxes onto all target locations. A box on a target becomes yellow.


Game Mechanics & Action Choice:
- The game is played on a grid using the following characters:
- To push a Box: If the Player is adjacent to a Box AND you want to move that Box, you can use
      actions:
 "up", "down", "left", "right" against the box to push it.
  The square beyond the box must be an empty Floor or empty Target.
- Player cannot push a Box into a Wall or another Box.
- The game is solved when all boxes are on target locations (all boxes are yellow).

Key Strategies:
1. Box pushing order: Think carefully about which boxes to push first.
2. Avoid deadlocks: Do not push boxes into corners (unless it is a target) or against walls
     where they cannot be moved out.
3. Player positioning: Position the player to push boxes effectively.

Here is the planner observation and its plan:
<image>

<place_holder_for_plan>

You should first describe the current position of the role, and output an action to achieve
     the plan.
Then describe the future position of the role at the next game state.
Finally you need to analyze whether the plan will be in progress, succeeded or failed after
     this action happens, or the plan is out-of-date.

Your response format MUST be:
Current role location: [Down/Up] [number], [Left/Right] [number] relative to the role position
      in plan image.
Move: [up/down/left/right]
Future role location: [Down/Up] [number], [Left/Right] [number] relative to the role position
     in plan image.
Plan [in progress/succeeded/failed/out of date]
```

Direct-action baseline prompt.

```
<image>
You are an intelligent AI player solving a Sokoban puzzle.
Your goal is to make strategic moves to push all boxes onto their designated target locations.
Sokoban Game Quick Guide:

Primary Goal: Push all brown boxes onto all target locations. A box on a target becomes yellow.


Game Mechanics & Action Choice:
- The game is played on a grid using the following characters:
- To push a Box: If the Player is adjacent to a Box AND you want to move that Box, you can use
      actions:
 "up", "down", "left", "right" against the box to push it.
  The square beyond the box must be an empty Floor or empty Target.
- Player cannot push a Box into a Wall or another Box.
- The game is solved when all boxes are on target locations (all boxes are yellow).

Key Strategies:
1. Box pushing order: Think carefully about which boxes to push first.
2. Avoid deadlocks: Do not push boxes into corners (unless it is a target) or against walls
     where they cannot be moved out.
3. Player positioning: Position the player to push boxes effectively.

Your response format MUST be:
Move: [up/down/left/right]

Example answer:
Move: up
```

Plan-as-CoT prompt.

```
You are an intelligent AI player solving a Sokoban puzzle.
```

```
Your goal is to make strategic moves to push all boxes onto their designated target locations.
Sokoban Game Quick Guide:

Primary Goal: Push all brown boxes onto all target locations. A box on a target becomes yellow.


Game Mechanics & Action Choice:
- The game is played on a grid using the following characters:
- To push a Box: If the Player is adjacent to a Box AND you want to move that Box, you can use
      actions:
 "up", "down", "left", "right" against the box to push it.
  The square beyond the box must be an empty Floor or empty Target.
- Player cannot push a Box into a Wall or another Box.
- The game is solved when all boxes are on target locations (all boxes are yellow).

Key Strategies:
1. Box pushing order: Think carefully about which boxes to push first.
2. Avoid deadlocks: Do not push boxes into corners (unless it is a target) or against walls
     where they cannot be moved out.
3. Player positioning: Position the player to push boxes effectively.

You should first give a plan, including the following two types:
Move to the location [Down/Up] [number], [Left/Right] [number] relative to the role.
or
Push [down/up/left/right] for [number] steps.

Then you should output an action to move or push box towards the planned position.

Your response format MUST be:
Plan: [plan]
Move: [up/down/left/right]

Example answer:
Plan: Move to the location Down 3, Left 4 relative to the role.
Move: down

here is the current observation:<image>
```

Plan-Chunk baseline prompt.

```
You are an intelligent AI player solving a Sokoban puzzle.
Your goal is to make strategic moves to push all boxes onto their designated target locations.
Sokoban Game Quick Guide:

Primary Goal: Push all brown boxes onto all target locations. A box on a target becomes yellow.


Game Mechanics & Action Choice:
- The game is played on a grid using the following characters:
- To push a Box: If the Player is adjacent to a Box AND you want to move that Box, you can use
      actions:
 "up", "down", "left", "right" against the box to push it.
  The square beyond the box must be an empty Floor or empty Target.
- Player cannot push a Box into a Wall or another Box.
- The game is solved when all boxes are on target locations (all boxes are yellow).

Key Strategies:
1. Box pushing order: Think carefully about which boxes to push first.
2. Avoid deadlocks: Do not push boxes into corners (unless it is a target) or against walls
     where they cannot be moved out.
3. Player positioning: Position the player to push boxes effectively.

You should first give a plan, including the following two types:
Move to the location [Down/Up] [number], [Left/Right] [number] relative to the role.
or
Push [down/up/left/right] for [number] steps.

Then you should output an action chunk to move or push box towards the planned position.

Your response format MUST be:
Plan: [plan]
Move: ([up/down/left/right], [up/down/left/right], ...)

Example answer:
Plan: Move to the location Down 3, Left 4 relative to the role.
Move: (down, down, down, left, left, left, left)

Here is the current observation:<image>
```

A.5.3 MINIGRID PROMPTS

Planner prompt.

```
<image>
You are controlling an agent in MiniGrid. The goal is to reach the green goal tile while
     avoiding walls and lava.
Some maps require picking up a key and toggling a door before reaching the goal.
The agent need to stand at the neighbor cell of the key/door and face to it to pickup/toggle
     it.
Analyze the current observation and output a short plan line with this exact format:

Plan: Go to [Down/Up] [number], [Left/Right] [number] relative to the agent; heading [north/
     east/south/west]; [pickup/toggle/arrive]

Example:
Plan: Go to Down 2, Left 3 relative to the agent; heading east; pickup
```

Actor-verifier prompt.

```
You are controlling an agent in MiniGrid. The goal is to reach the green goal tile while
     avoiding walls and lava.
Some maps require picking up a key and toggling a door before reaching the goal.
Allowed actions are: left, right, forward, pickup, toggle.
"left" and "right" make your orientation rotate, "forward" makes you go forward a cell along
     current orientation,
"pickup" pickups the key next to the agent in current orientation,
"toggle" opens the door next to the agent in current orientation.

Planner observation and plan:
<image>
{plan_line}

You need to analyze the current state first, then output the current action according to the
     plan, then predict the future state and plan status.

Your response format MUST be:
Current agent location: [Down/Up] [number], [Left/Right] [number] relative to the planner-
     image agent position
Current orientation: [north/east/south/west]
Move: [left/right/forward/pickup/toggle]
Future agent location: [Down/Up] [number], [Left/Right] [number] relative to the planner-image
      agent position
Future orientation: [north/east/south/west]
Plan [in progress/succeeded/out of date]

Here is the current observation:
<image>
```

Direct-action baseline prompt.

```
<image>
You are controlling an agent in MiniGrid.
Goal: reach the green goal tile while avoiding walls and lava.
Some maps require picking up a key and toggling a door before reaching the goal.
The agent need to stand at the neighbor cell of the key/door and face to it to pickup/toggle
     it.
Allowed actions: left, right, forward, pickup, toggle.

Your response format MUST be:
Move: [left/right/forward/pickup/toggle]

Example answer:
Move: forward
```

Plan-as-CoT prompt.

```
<image>
You are controlling an agent in MiniGrid. The goal is to reach the green goal tile while
     avoiding walls and lava.
Some maps require picking up a key and toggling a door before reaching the goal.
The agent need to stand at the neighbor cell of the key/door and face to it to pickup/toggle
     it.
Allowed actions are: left, right, forward, pickup, toggle.
"left" and "right" make your orientation rotate, "forward" makes you go forward a cell along
     current orientation,
```

```
"pickup" pickups the key next to the agent in current orientation,
"toggle" opens the door next to the agent in current orientation.

Analyze the current observation, output a short plan line and an action.

Answer format:
Plan: Go to [Down/Up] [number], [Left/Right] [number] relative to the agent; heading [north/
    east/south/west]; [pickup/toggle/arrive]
Move: [left/right/forward/pickup/toggle]

Example:
Plan: Go to Down 2, Left 3 relative to the agent; heading east; pickup
Move: forward
```

Plan-Chunk prompt.

```
<image>
You are controlling an agent in MiniGrid. The goal is to reach the green goal tile while
    avoiding walls and lava.
Some maps require picking up a key and toggling a door before reaching the goal.
The agent need to stand at the neighbor cell of the key/door and face to it to pickup/toggle
    it.
Allowed actions are: left, right, forward, pickup, toggle.
"left" and "right" make your orientation rotate, "forward" makes you go forward a cell along
    current orientation,
"pickup" pickups the key next to the agent in current orientation,
"toggle" opens the door next to the agent in current orientation.

Analyze the current observation, output a short plan line and an action list.

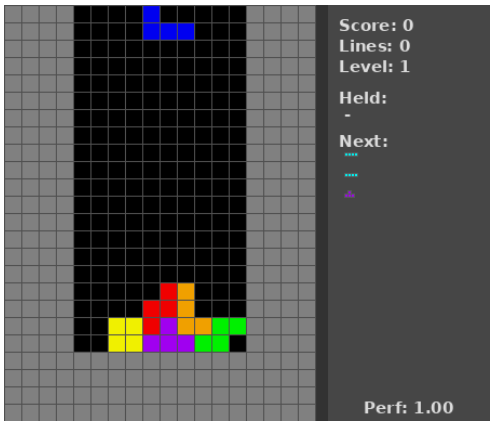
Answer format:
Plan: Go to [Down/Up] [number], [Left/Right] [number] relative to the agent; heading [north/
    east/south/west]; [pickup/toggle/arrive]
Move: ([left/right/forward/pickup/toggle], [left/right/forward/pickup/toggle], ...)

Example:
Plan: Go to Down 2, Left 3 relative to the agent; heading east; pickup
Move: (forward, forward, right, forward, forward, forward, pickup)
```

A.6 Qualitative Online Interaction

The following Tetris example is drawn from an actual online rollout.

Frame 16. Observe the image from the Tetris environment.



frame 16

Build planner prompt:

```
<image tokens of frame 16>Analyze this Tetris game state and plan the optimal placement.
    Tetris Game Mechanics:

1. The game board is a grid where pieces (tetrominoes) fall from the top.
2. Your goal is to clear lines by filling all cells in a row, which removes that row.
```

```
4. The game ends if the pieces stack up to the top of the board.

Here is the recommanded policy to play the game: Keep the board flat and avoid tall stacks.
Prefer filling holes and completing lines. Rotate only when it improves fit.
You should answer with the following format:

Optimal Placement: column [number], row [number], [keep current orientation/rotate left/rotate
      right/rotate 180 degrees]
The block coordinates is defined as follows:
start from bottom left, start from 1, column defined by leftest unit of the block, row defined
      by bottom unit of the block.
```

⟶

Model response (planner):

```
Optimal Placement: column 1, row 1, rotate left
```

⟶

Use the current observation (frame 16) and the planner observation (frame 16).

Build actor-verifier prompt:

```
<image tokens of frame 16>Analyze this Tetris game state and output the optimal next move
     according to the planner generated optimal placement plan, then verify whether the plan
     status. Tetris Game Mechanics:

1. The game board is a grid where pieces (tetrominoes) fall from the top.
2. Your goal is to clear lines by filling all cells in a row, which removes that row.
4. The game ends if the pieces stack up to the top of the board.

Available Actions (Format: move: (action_name_from_list, frame_count) where frame_count is
     1-10 and action_name_from_list MUST be one of the following exact strings):
- "no_op": Do nothing for this step.
- "left": Move the current piece one cell to the left.
- "right": Move the current piece one cell to the right.
- "rotate_left": Rotate the current piece 90 degrees counter-clockwise.
- "rotate_right": Rotate the current piece 90 degrees clockwise.
- "soft_drop": Move the current piece one cell down.
- "hard_drop": Instantly drop the current piece to its lowest possible position.

Here is the planner observation and its plan:
<image tokens of frame 16>
Optimal Placement: column 1, row 1, rotate left
The block coordinates is defined as follows:
start from bottom left, start from 1, column defined by leftest unit of the block, row defined
      by bottom unit of the block.
You should first decribe the current position and orientation of the falling block and output
     an action to move towards the planned optimal placement.
Then decribe the future position and orientation of the falling block at the next game state.
Finally you need to analyze whether the plan will be in progress, succeeded, failed, or out of
      date after this action happens.
You should answer with the following format:
Current state: column [number], row [number], orientation [0/1/2/3]
move: (action_name_from_list, frame_count)
Future state: column [number], row [number], orientation [0/1/2/3]
Plan [in progress/succeeded/failed/out of date]
```

⟶

Model response (actor-verifier):

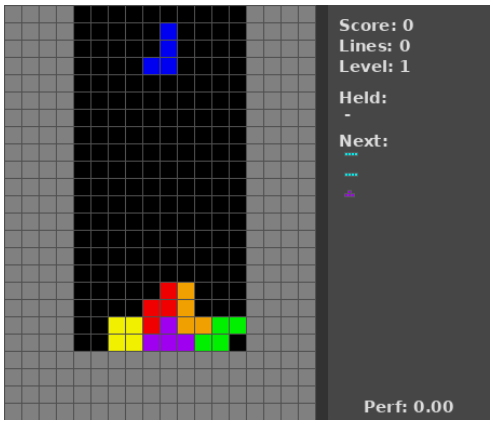
```
Current state: column 5, row 19, orientation 0
move: (rotate_left, 1)
Future state: column 5, row 17, orientation 3
Plan in progress
```

⟶

Parse action and verification status: action `(rotate_left, 1)`; normalized action `rotate_left`; plan status `in progress`. The predicted plan status is `in progress`. Execute `(rotate_left, 1)` and observe frame 17; retain the current plan for the next actor-verifier call.

⟶

Frame 17. Observe the image from the Tetris environment.



frame 17

Use the current observation (frame 17) and the planner observation (frame 16).

Build actor-verifier prompt:

```
<image tokens of frame 17>
...
Here is the planner observation and its plan:
<image tokens of frame 16>
Optimal Placement: column 1, row 1, rotate left
...
```

⟶

Model response (actor-verifier):

```
Current state: column 5, row 17, orientation 3
move: (left, 1)
Future state: column 4, row 16, orientation 3
Plan in progress
```

⟶

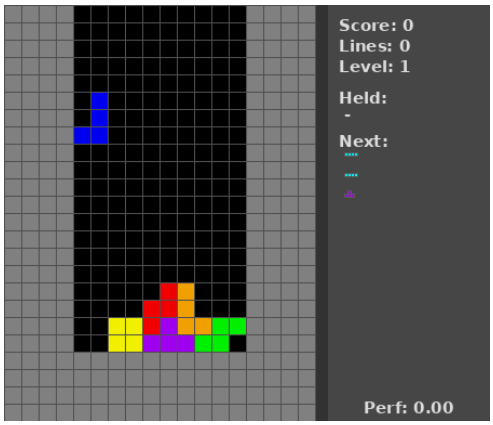
Parse action and verification status: action `(left, 1)`; normalized action `left`; plan status `in progress`. The predicted plan status is `in progress`. Execute `(left, 1)` and observe frame 18; retain the current plan for the next actor-verifier call.

⟶

. . .

Intermediate calls at frames 18–20 omitted. After the intervening actions, observe frame 21.

Frame 21. Observe the image from the Tetris environment.



frame 21

Use the current observation (frame 21) and the planner observation (frame 16).

Build actor-verifier prompt:

```
<image tokens of frame 21>
...
Here is the planner observation and its plan:
<image tokens of frame 16>
Optimal Placement: column 1, row 1, rotate left
...
```

⟶

Model response (actor-verifier):

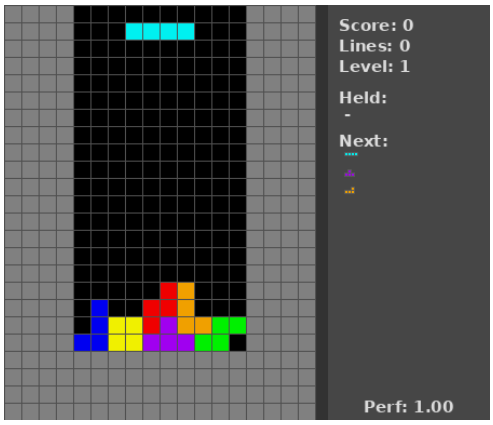
```
Current state: column 1, row 13, orientation 3
move: (hard_drop, 1)
Future state: column 1, row 1, orientation 3
Plan succeeded
```

⟶

Parse action and verification status: action `(hard_drop, 1)`; normalized action `hard_drop`; plan status `succeeded`. The predicted plan status is `succeeded` (completed). Execute `(hard_drop, 1)`, observe frame 22, and build a new planner prompt.

⟶

Frame 22. Observe the image from the Tetris environment.



frame 22

Build planner prompt:

```
<image tokens of frame 22>
...
```

⟶

Model response (planner):

```
Optimal Placement: column 1, row 2, rotate right
```

⟶

. . .