# THINK IN LANGUAGE, ACT IN CODE: SEARCHING PROGRAMMATIC POLICIES VIA NATURAL LANGUAGE

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Programmatic policies offer an appealing representation for reinforcement learning because they can translate the general knowledge encoded in foundation models into functions that issue low-level actions in the environment. However, synthesizing effective programs through interaction with an environment is challenging, as the space of possible programs is vast and small syntactic changes can lead to large and often unpredictable changes in behavior. We tackle this challenge by searching in an abstraction of the program space through LLM-proposed natural language programs, using feedback from environment interactions to guide the search. Each natural language program specifies a candidate strategy and is translated into an executable policy through iterative Python code synthesis. This creates a two-level search process in which natural language guides the exploration of strategies while program synthesis grounds these strategies into executable policies. Experiments on a large set of MiniHack tasks show that searching through natural language programs improves both the efficiency of programmatic policy search and the quality of the policies discovered.

## 1 INTRODUCTION

Reinforcement learning (RL) has achieved strong performance in challenging sequential decision-making problems, but the resulting policies are often represented by large neural networks that can be difficult to inspect and modify (Sutton & Barto, 2018; Mnih et al., 2015; Schulman et al., 2017). Programmatic policies provide a complementary representation: the policy is an executable program that maps observations to actions. Earlier work has synthesized programmatic policies using domain-specific languages, imitation from neural policies, state machines, learned program representations, and direct search (Bastani et al., 2018; Verma et al., 2018; Inala et al., 2020; Trivedi et al., 2021; Qiu & Zhu, 2022; Liu et al., 2023b; Abdollahi et al., 2023). Such policies expose their control flow explicitly, making them amenable to direct inspection and modification.

Large language models (LLMs) substantially expand the range of programs that can be synthesized by generating executable programs from natural-language specifications (Chen et al., 2021; Liang et al., 2023). Recent work has used LLMs as proposal mechanisms within iterative optimization loops, generating candidate programs that are evaluated and subsequently improved based on feedback (Romera-Paredes et al., 2024; Liu et al., 2024; Ma et al., 2024b; Cheng et al., 2024; Ishida et al., 2024; Novikov et al., 2025). These capabilities suggest an alternative approach to policy discovery: instead of optimizing neural parameters, use an LLM to propose policies, evaluate them through interaction with the environment, and use the resulting feedback to guide subsequent proposals.

The central difficulty is search. The space of Python policies is enormous and non-differentiable, and small changes to a program can produce either negligible or substantial changes in its behavior. Pure score-guided program search may therefore spend substantial computation exploring low-level implementation variations instead of meaningful changes to the underlying policy strategy. Execution-guided refinement provides richer feedback by exposing trajectories and failures to the optimizer (Cheng et al., 2024; Kuang et al., 2025; Ishida et al., 2024), but repeatedly refining a single program may hinder the exploration of behaviorally distinct policies.

Natural language offers a useful abstraction of this search space. Instead of searching only over executable code, one can represent a policy's intended behavior as a *natural-language policy (NLP)* and organize exploration around changes to that NLP. We call search over such NLPs *semantic search*, in



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.

contrast to search over their concrete code realizations. Evolution of Heuristics (EoH), for example, co-evolves natural-language "thoughts" and executable heuristics (Liu et al., 2024), while MLES extends this idea to programmatic control policies and incorporates behavioral evidence into evolutionary policy discovery (Hu et al., 2026). More broadly, Tree of Thoughts, RAP, and Language Agent Tree Search demonstrate the value of searching over language-level reasoning states rather than committing to one reasoning trajectory (Yao et al., 2023a; Hao et al., 2023; Zhou et al., 2024).

We argue that efficient semantic search requires more than attaching a natural-language rationale to each executable program. First, an NLP should be structured enough to support localized modifications, enabling search in the natural-language space. Second, it should be grounded in observed behavior, allowing the search to determine which parts were successfully executed and where execution deviated from the intended behavior. Third, a promising NLP should not be discarded simply because its first code realization is defective: the quality of a strategy represented by an NLP and the quality of a particular code realization are distinct. Finally, knowledge acquired through experience while exploring one NLP should be reusable when exploring others rather than repeatedly rediscovered.

We introduce TACO (**T**hink in Language, **A**ct in **CO**de), a framework for programmatic policy discovery that searches a tree of natural-language policies. Each NLP includes a description of the candidate policy strategy together with a plan for its intended behavior. After a new NLP is proposed, an LLM realizes it as Python code conditioned on the NLP. The code is executed in the environment, and the resulting transitions are used both to assess whether the code faithfully realizes the NLP and to guide subsequent code refinements. Instead of assigning the NLP node the score of a single code sample, TACO uses *committed code realization*: it keeps the NLP fixed for a bounded number of code-refinement steps while improving its executable realization. A UCB-style selection rule (Auer et al., 2002; Kocsis & Szepesvári, 2006) allocates search effort among NLP nodes, while a shared natural-language knowledge library makes information about environment dynamics, object semantics, and recurring failure modes available across branches. This design creates two coupled search levels. The outer search explores *what to do* by searching over structured NLPs, while the inner code-realization process determines *how to realize it* as executable code.

Our contributions are: **(1)** We formulate programmatic policy discovery as *structured semantic search*, in which natural-language policies are explicit search objects representing intended policy behavior rather than auxiliary explanations attached to code. **(2)** We introduce *committed code realization*, in which each NLP receives a bounded code-refinement budget, allowing the strategy to be evaluated through multiple code-refinement steps rather than a single stochastic code realization. **(3)** We combine semantic tree search with a shared natural-language knowledge library, allowing execution-derived discoveries to inform the exploration and realization of policies across different branches. **(4)** We evaluate TACO on a broad collection of MiniHack tasks (Samvelyan et al., 2021), comparing against code-level optimization and semantic-search baselines and ablating the mechanisms above.

## 2 Problem Definition

We consider a Markov decision process (MDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $P(s_{t+1} \mid s_t, a_t)$ is the transition distribution, $R(s_t, a_t, s_{t+1})$ is the reward function, and $\gamma \in [0, 1]$ is the discount factor. At time step $t$, the agent observes $s_t \in \mathcal{S}$, selects $a_t \in \mathcal{A}$, and receives $r_t = R(s_t, a_t, s_{t+1})$; the environment then transitions to $s_{t+1} \sim P(\cdot \mid s_t, a_t)$.

Let $\mathcal{L}$ denote a programming language and $\Pi_{\mathcal{L}}$ the space of executable policy programs expressible in $\mathcal{L}$. The language may be a domain-specific language or a general-purpose language such as Python; in this work, we use Python. Each program $\pi \in \Pi_{\mathcal{L}}$ encoding a policy, when executed on a state $s_t$, produces an action $a_t = \pi(s_t)$, and therefore induces a deterministic policy from $\mathcal{S}$ to $\mathcal{A}$.

Executing $\pi$ in a finite-horizon episodic environment induces a distribution over trajectories $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_T)$, which we denote by $p(\tau \mid \pi)$. The quality of a program is measured by its expected discounted return, $J(\pi) = \mathbb{E}_{\tau \sim p(\tau|\pi)} \sum_{t=0}^{T-1} \gamma^t r_t$. Our goal is to find an executable programmatic policy $\pi^\star \in \Pi_{\mathcal{L}}$ that maximizes the expected discounted return: $\pi^\star \in \arg\max_{\pi \in \Pi_{\mathcal{L}}} J(\pi)$.

# 3 STRUCTURED SEMANTIC SEARCH FOR PROGRAMMATIC POLICIES

Directly searching $\Pi_{\mathcal{L}}$ is difficult because small syntactic changes to a program can correspond to large behavioral changes, while programs with substantially different syntax can implement similar high-level strategies. We therefore introduce a semantic policy representation in natural language. Let $\psi \in \Psi$ denote a *natural-language policy* (NLP), where $\Psi$ is the space of natural-language policies describing candidate policy strategies. An NLP specifies *what* behavior a policy should implement without fixing the exact program used to implement it.

An NLP can admit multiple executable realizations. Let $\Pi_\psi \subseteq \Pi_{\mathcal{L}}$ denote the set of executable programs that realize NLP $\psi$. Different programs in $\Pi_\psi$ may realize the same semantic strategy with different degrees of fidelity and achieve different returns. For a fixed NLP $\psi$, the code-realization objective is therefore to find its best executable realization, $\pi^\star_\psi \in \arg\max_{\pi \in \Pi_\psi} J(\pi)$. This corresponds to the inner code-realization problem: improving the executable policy while keeping its NLP fixed. We associate each NLP with the return of its best executable realization, $V(\psi) = J(\pi^\star_\psi)$. The semantic-level objective is then to identify $\psi^\star \in \arg\max_{\psi \in \Psi} V(\psi)$, which corresponds to the outer search problem over candidate policy strategies. Thus, the overall problem consists of two coupled levels: semantic search for effective NLPs and code realization for obtaining high-quality executable realizations through which those NLPs can be evaluated.

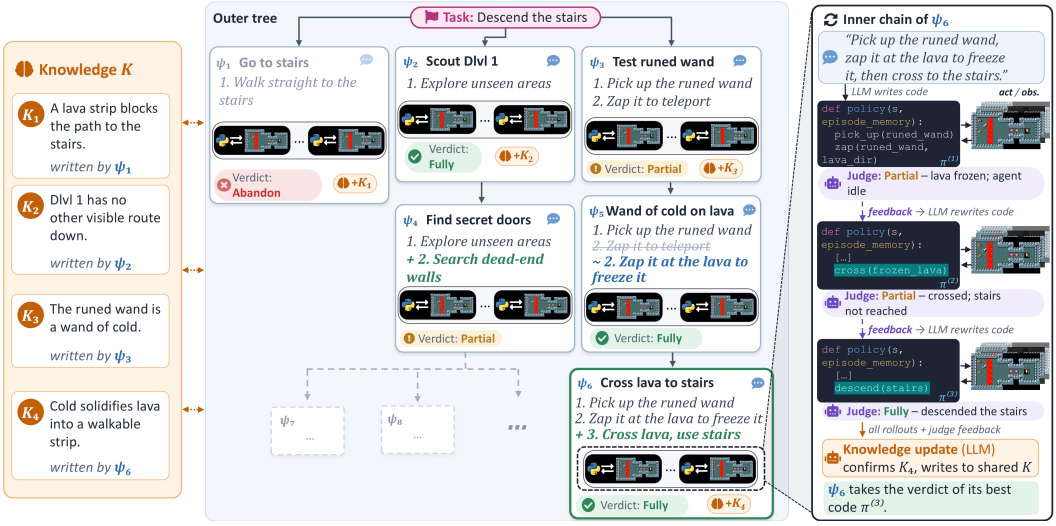



Figure 1: **TACO overview. (Middle) Outer tree:** plans $\psi$ in natural language are expanded by editing a parent plan. Each plan is executed and assigned a verdict by an LLM judge in its refinement chain. **(Left) Knowledge $\mathcal{K}$:** facts discovered by any node are written to a shared knowledge base that informs all subsequent planning. **(Right) Inner chain:** each plan is compiled into a code policy $\pi$, which is iteratively rewritten using judge feedback on its rollouts. A knowledge-update step then distills the rollouts into new entries in $\mathcal{K}$, and the node inherits the verdict of its best policy.

## 3.1 OVERVIEW

We address the coupled semantic and implementation-level search problems with TACO, a two-level framework for programmatic policy discovery. The outer-level search explores NLPs $\psi \in \Psi$ using a semantic tree (Section 3.2), approximating the search for $\psi^\star$. The inner-level search operates on $\psi$ and iteratively refines its executable realization, approximating the search for $\pi^\star_\psi$ (Section 3.3).

Figure 1 illustrates the interaction between these two levels. Each node in the semantic tree represents an NLP and maintains a sequence of executable realizations. When the outer search expands a node, it generates a child NLP and invokes the inner realization process. The inner process performs *committed code realization*, keeping the NLP fixed for a bounded budget of $M$ code-refinement

steps while using environment interaction to evaluate and improve its executable realization. Following the LLM-as-a-Judge paradigm (Zheng et al., 2023), an LLM determines whether the NLP has been fully implemented, remains partially implemented, or should be abandoned. This assessment subsequently determines whether the node can participate in further search and, if so, how the outer search can expand it. We describe each of these components in this section.

Environment interaction also produces reusable knowledge about the environment. This knowledge is accumulated in a shared library $\mathcal{K}$ and is available to both levels of search, allowing information discovered while realizing one NLP to inform subsequent code refinement and semantic expansion elsewhere in the tree. We estimate the expected return $J(\pi)$ by executing $\pi$ for $E$ episodes.

### 3.2 Outer-Level Semantic Search

The outer level of TACO searches over NLPs using a semantic tree. Each node $N$ represents an NLP $\psi_N$, and each edge represents a semantic update that produces a new NLP from its parent. Multiple examples of learned semantic trees are provided in Appendix D.

**NLP representation.** An NLP is represented as $\psi = \left(d, (p_i)_{i=1}^{I}\right)$, where the *description* $d$ describes the high-level rationale of the policy, and $(p_i)_{i=1}^{I}$ is the *plan* that specifies its intended behavior as an ordered sequence of stages. Each node $N$ stores its NLP $\psi_N$, its best executable realization $\pi_N^*$, and a realization verdict $v_N \in \{\texttt{abandon}, \texttt{partial}, \texttt{full}\}$ coming from the LLM judge.

**Selection.** Starting from the root, TACO recursively descends the semantic tree. Progressive widening controls when additional nodes can be added to the tree (Couëtoux et al., 2011). A child is appended to node $N$ when $|\text{children}(N)| < \lceil k\sqrt{n_N} \rceil$. Here, $n_N$ is the visit count of node $N$ and $k$ is a hyperparameter that controls the rate of expansion. When $N$ is not eligible for further expansion, search descends to the non-abandoned child $N'$ maximizing

$$\text{UCB}(N') = Q_{N'} + c_{\text{ucb}} \sqrt{\frac{\ln n_N}{1 + n_{N'}}}, \tag{1}$$

where $Q_{N'}$ summarizes the observed quality of the subtree rooted at $N'$, $n_{N'}$ is its visit count, and $c_{\text{ucb}}$ controls exploration (Auer et al., 2002; Kocsis & Szepesvári, 2006).

**Verdict-conditioned expansion.** Once a node $N$ is selected for expansion, its realization verdict determines how the LLM produces the new child's NLP. Nodes with $v_N = \texttt{abandon}$ are never selected for expansion. If $v_N = \texttt{partial}$, the LLM modifies the existing stages of the plan. If $v_N = \texttt{full}$, the existing stages are preserved, and the LLM adds new stages to the plan. In both cases, the child's NLP differs from its parent's in a few stages, so each edge of the semantic tree corresponds to a *local* update of the plan. The child $N'$ is generated by

$$\psi_{N'} = \text{LLM}_{\text{nlp}}\left(\psi_N, v_N, \Psi_N^{\text{child}}, \mathcal{K}\right),$$

where $\Psi_N^{\text{child}}$ contains the previously generated children of $N$. Providing previous children discourages repeated semantic updates, while shared knowledge $\mathcal{K}$ allows findings acquired through execution elsewhere in the tree to inform new proposals. The resulting NLP $\psi_{N'}$ is passed to the inner-level realization process described in Section 3.3.

**Backpropagation.** After the inner-level realization process for a node $N$ terminates, its search value is determined by the empirical return of its best executable realization, $\hat{J}(\pi_N^*)$. We normalize this return using the range of returns observed during the search: $\tilde{r}_N = \frac{\hat{J}(\pi_N^*) - R_{\min}}{R_{\max} - R_{\min}}$.

The normalized return is then backed up through $N$ and its ancestors. For each node $A$, we update

$$S_A \leftarrow S_A + \tilde{r}_N, \qquad n_A \leftarrow n_A + 1, \qquad Q_A \leftarrow \frac{S_A}{n_A}.$$

Thus, $Q_A$ is the mean normalized return of resolved nodes in the subtree rooted at $A$ and provides the exploitation term in Eq. 1. The values of $Q$ are initialized to zero.

### 3.3 INNER-LEVEL CODE REALIZATION

Each newly created NLP node is assigned an inner-level realization process that searches for an effective executable implementation while keeping its NLP fixed. Rather than evaluating an NLP from a single code-generation attempt, TACO commits to it for up to $M$ refinement steps, allowing execution feedback to guide successive implementations.

**Committed code realization.** Let $N$ be a newly created node with parent $P$. The realization process is initialized from the parent's best executable policy. Each refinement step $t \in \{0, \dots, M-1\}$ starts by executing the current policy $\pi_N^t$ in the environment to collect $E$ trajectories $\mathcal{D}_N^t = \{\tau_i^t\}_{i=1}^E$. These are added to the node's local refinement history, $H_N^{t+1} = H_N^t \cup \{\mathcal{D}_N^t\}$ with $H_N^0 = \emptyset$. The code-generation LLM then produces the next policy from the NLP, the current code, the history, and the shared knowledge library $\mathcal{K}$:

$$\pi_N^0 = \pi_P^*, \qquad \pi_N^{t+1} = \mathrm{LLM}_{\mathrm{code}}\big(\psi_N, \pi_N^t, H_N^{t+1}, \mathcal{K}\big). \tag{2}$$

Refinement proceeds from the most recent realization, even when an update reduces performance. In parallel, the node maintains the best executable realization $\pi_N^*$ encountered so far. Among non-crashing realizations, policies are compared by empirical return $\hat{J}(\pi)$, with ties resolved in favor of the more recent realization. The resulting $\pi_N^*$ determines the value of the node.

**Realization assessment.** At each refinement, an LLM judge evaluates whether the current code realizes the behavioral stages of $\psi_N$. Given the NLP, the corresponding execution evidence, and the shared knowledge library, the judge returns

$$v_N^t = \mathrm{LLM}_{\mathrm{judge}}(\psi_N, \mathcal{D}_N^t, \mathcal{K}),$$

where $v_N^t \in \{\texttt{abandon}, \texttt{partial}, \texttt{full}\}$ denotes the realization result at refinement step $t$. The judge thus evaluates implementation fidelity from observed execution (Zheng et al., 2023).

At each refinement step $t$, the judge produces a verdict $v_N^t$. A verdict of `full` or `abandon` terminates refinement early, with the corresponding verdict assigned as the final node-level outcome, i.e., $v_N = v_N^t$. Otherwise, refinement continues until the budget of $M$ steps is exhausted. If no terminating verdict is reached within this budget, the final outcome is set to $v_N = \texttt{partial}$.

**Knowledge extraction.** At the end of code realization, a separate LLM call extracts reusable natural-language knowledge from the node's local refinement history $H_N$, which is the latest local refinement history of node $N$:

$$\Delta\mathcal{K}_N = \mathrm{LLM}_{\mathrm{know}}(H_N, \mathcal{K}), \qquad \mathcal{K} \leftarrow \mathcal{K} \cup \Delta\mathcal{K}_N.$$

The shared library captures environment-level information such as observed dynamics, object semantics, and recurring failure modes. Knowledge acquired while realizing one NLP can therefore inform both later code refinements and semantic expansions in other branches of the tree.

## 4 EXPERIMENTS

**Environments.** We evaluate on five MiniHack environments: QUEST-EASY, WAND-OF-DEATH-HARD, CORRIDOR-R3, KEY-ROOM-S15, and CORRIDOR-BATTLE, spanning navigation, item interaction, and combat. Instead of using restricted task-specific action spaces, we use the much more challenging original NLE action space of 121 actions (Küttler et al., 2020). Rewards are sparse and terminal: successful completion yields 1 and all other transitions yield 0. This means that a $\hat{J}(\pi)$-value of 1.0 indicates that the agent solves the problem.

**Baselines.** We compare against two closely related LLM-based approaches. **MLES-UCB** adapts MLES (Hu et al., 2026) by replacing its evolutionary search with the same progressive-widening and UCB-based tree-search mechanism used by TACO, controlling for the choice of high-level search mechanism. **GCO** follows (Kuang et al., 2025), directly refining executable Python policies from environment feedback without a separate semantic search over NLPs.

**Language models.** We instantiate $\text{LLM}_{\text{nlp}}$ with DeepSeek-V4-Flash (DeepSeek-AI et al., 2026) and $\text{LLM}_{\text{judge}}$ with Gemini-3.1-Flash-Lite. We use a temperature of 0.7 and a maximum output length of 64K tokens. Equivalent LLM operations use the same models across methods. Complete prompts and model configurations are provided in Appendix C.

**Evaluation protocol.** We perform 10 independent runs on QUEST-EASY, WAND-OF-DEATH-HARD, and KEY-ROOM-S15, and 5 on CORRIDOR-R3 and CORRIDOR-BATTLE. Each program is evaluated over a fixed, environment-dependent number of episodes reported in Appendix B. Following our program-discovery objective, we report the performance of the best program found up to each point in the search. We measure search cost by both cumulative environment interactions and cumulative LLM output tokens. The latter includes all generated outputs, including policy specifications, executable programs, extracted knowledge, and realization assessments. The curves in the plots report the mean across independent runs, with shading indicating $\pm 1$ standard deviation.

**Search configuration.** For TACO, we allow up to $M = 5$ code-refinement steps per NLP and set progressive widening coefficient $k = 0.7$ and UCB exploration coefficient $c_{\text{ucb}} = \sqrt{2}$. These hyperparameters are fixed across environments; complete settings for all methods are reported in Appendix B.

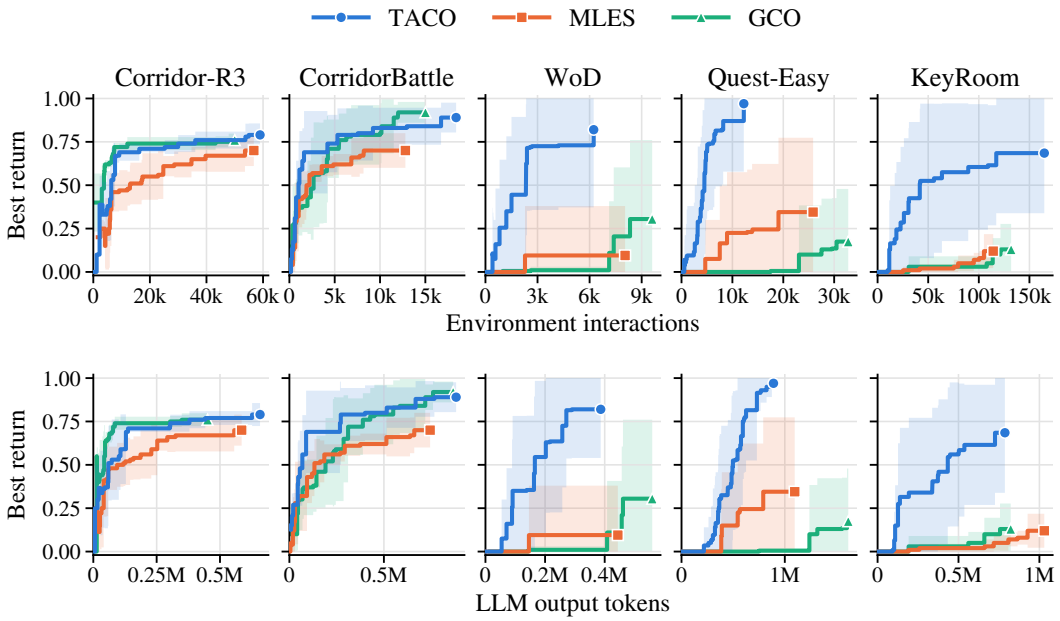



Figure 2: Best program performance throughout search on five MiniHack environments. The top row reports performance as a function of cumulative environment interactions, while the bottom row reports performance as a function of cumulative LLM output tokens. Curves show the mean across independent runs and shaded regions indicate $\pm 1$ std.

**Results.** Figure 2 compares the best program discovered by each method. The difference is particularly pronounced on WAND-OF-DEATH-HARD, QUEST-EASY, and KEY-ROOM-S15, where TACO reaches substantially higher success rates while MLES-UCB and GCO frequently remain at low performance. These environments admit a broader range of interactions and require discovering appropriate combinations of behaviors. For example, WAND-OF-DEATH-HARD requires identifying an effective use of the wand against the troll, KEY-ROOM-S15 requires exploring to locate the key and using it appropriately, and QUEST-EASY requires coordinating several behaviors, including traversing lava and fighting monsters. The stronger performance of TACO on these tasks is consistent with its semantic search exploring this broader space of behaviors more effectively. On CORRIDOR-R3 and CORRIDOR-BATTLE, all methods discover effective policies. The same qualitative pattern holds whether performance is plotted against environment interactions or LLM output tokens, so the gains of TACO do not arise simply from generating more LLM output.

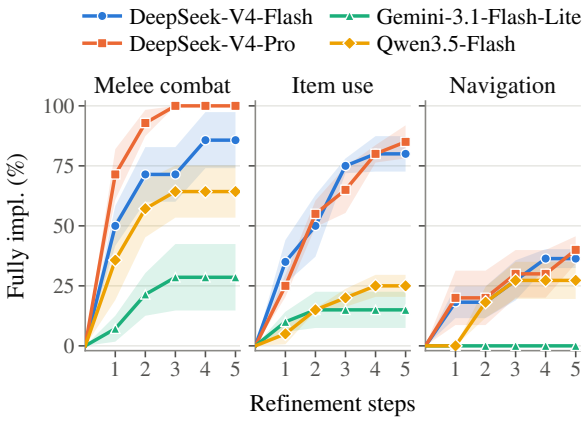



Figure 3: Percentage of nodes labeled *full* across refinement steps for four LLMs on 45 NLPs, grouped by behavioral category: melee combat (14), item use (20), and navigation (11). Lines show group means and shaded regions indicate standard deviation across NLPs.

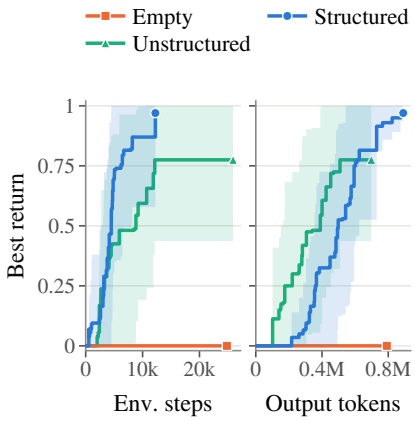



Figure 4: Best program performance on QUEST-EASY using empty, unstructured, and structured NLPs, versus environment steps and LLM output tokens. Lines show mean $\pm 1$ std over 10 seeds.

## 5 WHAT MAKES SEMANTIC SEARCH EFFECTIVE?

The experiment results in Figure 2 show that TACO significantly improves program discovery on challenging MiniHack tasks. We next examine the components that make this search effective.

### 5.1 COMMITTED CODE REALIZATION

We study whether committing to multiple code-refinement steps is necessary to realize NLPs. We run TACO with a maximum refinement budget of $M = 5$ on QUEST-EASY and CORRIDOR-BATTLE, using DeepSeek-V4-Flash, DeepSeek-V4-Pro, Gemini-3.1-Flash-Lite, and Qwen3.5-Flash as code-generation LLMs. From the resulting search trees, we group NLP nodes into three recurring behavioral categories—melee combat, item use, and navigation—and measure the cumulative fraction classified as `full` after each refinement step.

Figure 3 shows that code realization improves substantially with additional refinement, although the rate varies across behaviors and LLMs. For example, with DeepSeek models on "Item use", fully realized NLPs increase from roughly 25% after one step to 75% after five. Additional refinement thus yields more faithful realizations and therefore more accurate value estimates for the outer semantic search, motivating our committed code-realization procedure.

### 5.2 ADAPTIVE REALIZATION ASSESSMENT

Committed realization allows multiple refinement opportunities, but not every NLP requires the same amount of refinement. TACO therefore uses a realization judge after each refinement step: realizations classified as `full` or `abandon` terminate early, while those classified as `partial` continue until receiving a terminal outcome or reaching the maximum budget $M$.

We evaluate the effect of this mechanism on QUEST-EASY by running TACO with and without the realization judge for $M \in \{1, 3, 5, 10\}$, keeping all other components unchanged. Figure 5 (left) shows that the judge substantially improves the best return. Without the judge, larger refinement budgets provide little benefit, while adaptive assessment substantially improves returns.

Figure 5 (right) shows how the judge allocates this refinement budget. Across $M \in \{3, 5, 10\}$, many NLPs receive a terminal *finished* or *abandoned* outcome within the first few refinement steps, while only a small fraction uses the full budget. The judge therefore allocates refinement adaptively instead of applying the maximum number of steps uniformly to every NLP. Together with the results shown in Section 5.1, these results show that multiple refinement opportunities are useful, but deciding when to stop refinement is also important for effective semantic search.

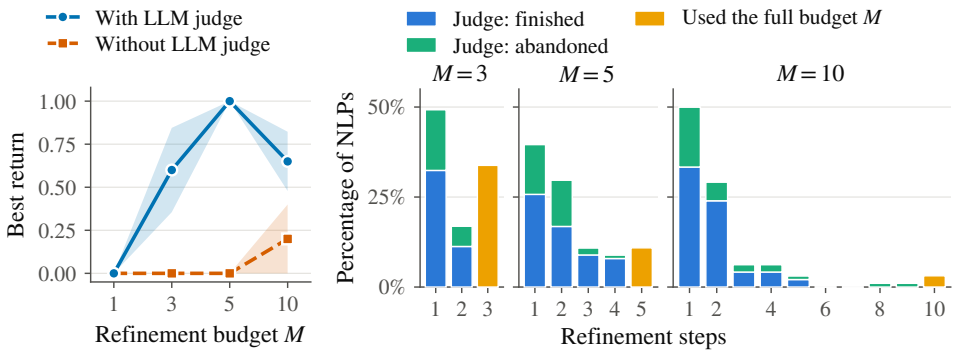



Figure 5: Effect of adaptive realization assessment on QUEST-EASY. **Left:** Best return with and without the realization judge as the maximum refinement budget $M$ increases; shaded regions show $\pm 1$ std over 5 seeds. **Right:** Percentage of NLPs receiving a terminal judge outcome (*finished* or *abandoned*) at each refinement step; orange marks NLPs that use the full budget.

### 5.3 STRUCTURED NLP REPRESENTATION

The semantic search in TACO represents each NLP as a structured specification containing a description and behavioral plan. We evaluate the importance of this representation on QUEST-EASY using three variants: *structured*, which uses the full NLP representation with description and staged plan; *unstructured*, which represents the policy strategy as unconstrained natural language; and *empty*, which provides no NLP specification.

Figure 4 shows that both NLP representations substantially outperform the empty variant. Structured NLPs ultimately reach near-optimal performance, while unstructured NLPs plateau at a lower return. This pattern holds for both environment interactions and LLM output tokens, although unstructured NLPs initially improve faster in token efficiency.

These results show that an explicit NLP representation is vital for semantic search, and that structuring this representation into staged plans further improves the quality of the policies discovered.

### 5.4 HOW DOES SHARED KNOWLEDGE HELP?

Finally, we examine the effect of the shared knowledge library $\mathcal{K}$. Since $\mathcal{K}$ is available during both code realization and semantic search, we analyze its effect separately on QUEST-EASY.

We first isolate code realization by taking NLPs generated during search with knowledge and resynthesizing them without access to $\mathcal{K}$, using the same code-generation LLM, starting policy, and a budget of five refinement steps. Table 1a shows access to $\mathcal{K}$ has no significant effect on realization success, the number of required refinement steps, or token usage.

Next we evaluate the impact of the knowledge library $\mathcal{K}$ on the complete semantic search on QUEST-EASY. As shown in Table 1b, access to $\mathcal{K}$ more than doubles the fraction of generated NLPs that are fully realized, from $0.07$ to $0.15$. Among runs that solve the task, it also reduces the number of generated NLPs before discovering the first successful policy from $11.83$ to $8.60$, while the total number of generated NLPs does not differ significantly. These results suggest that the main benefit of shared knowledge is in guiding semantic search toward NLPs that are more likely to yield successful realizations, rather than uniformly making a fixed NLP easier to implement.

## 6 RELATED WORK

**Programmatic policies and LLM-guided program search.** Programmatic RL represents policies as executable programs and searches for them through distillation, program synthesis, or learned search spaces (Bastani et al., 2018; Verma et al., 2018; Inala et al., 2020; Trivedi et al., 2021; Qiu & Zhu, 2022; Liu et al., 2023b; Abdollahi et al., 2023). More recently, LLMs have been used to propose and optimize executable programs through evolutionary or execution-guided search (Romera-

(a) Code realization on QUEST-EASY.

| Metric | w/o $\mathcal{K}$ | w/ $\mathcal{K}$ |
|---|---|---|
| Fully implemented (%) | $84.8 \pm 18.9$ | $77.4 \pm 4.6$ |
| # Required refinement steps | $2.63 \pm 0.48$ | $2.71 \pm 0.44$ |
| Output tokens per NLP (k) | $60.9 \pm 9.4$ | $56.2 \pm 10.4$ |

(b) Semantic search on QUEST-EASY.

| Metric | w/o $\mathcal{K}$ | w/ $\mathcal{K}$ |
|---|---|---|
| # generated NLPs | $13.62 \pm 4.60$ | $11.10 \pm 3.63$ |
| Fully realized NLP rate | $0.07 \pm 0.07$ | $0.15 \pm 0.09$ |
| NLPs to first success$^{\dagger}$ | $11.83 \pm 4.36$ | $8.60 \pm 1.51$ |

Table 1: Effect of the shared knowledge library $\mathcal{K}$ on code realization and semantic search. **Left**: re-synthesis of 31 fixed NLPs with and without $\mathcal{K}$. **Right**: semantic search with and without $\mathcal{K}$ over 10 seeds. Values are mean $\pm$ std. $^{\dagger}$NLPs generated before first reaching return 1.0.

Paredes et al., 2024; Liu et al., 2024; Novikov et al., 2025; Ma et al., 2024b; Ishida et al., 2024; Kuang et al., 2025). Our work instead makes a natural-language description of a complete policy strategy the primary object of search, while executable programs serve as locally refined realizations used to evaluate semantic candidates.

**Natural-language search and planning.** Natural language has also been used as a search representation for reasoning and planning. Tree of Thoughts, RAP, and Language Agent Tree Search perform search over language-level reasoning states, partial solutions, or plans (Yao et al., 2023a; Hao et al., 2023; Zhou et al., 2024), while interactive agents use language reasoning together with environment feedback (Yao et al., 2023b; Huang et al., 2023; Sun et al., 2023). In contrast, each node in our semantic search represents a complete, persistent policy specification whose quality is established by realizing and evaluating an executable policy in the environment.

**Semantic evolution and experience reuse.** Most closely related to our work, EoH jointly evolves natural-language heuristic ideas and executable implementations (Liu et al., 2024), while MLES represents programmatic-control candidates using code, natural-language thoughts, metrics, and behavioral evidence (Hu et al., 2026). Our framework explicitly separates global semantic search from local code realization: NLPs are the objects of tree search, while executable programs are locally refined realizations. Execution-derived knowledge is additionally shared across the semantic tree, similar to experience-reuse mechanisms in language agents (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2024). We provide a broader discussion of related work in Appendix A.

## 7 CONCLUSION

We introduced TACO, a framework for programmatic policy discovery that separates the search for *what a policy should do* from the problem of *how that behavior should be implemented*. Instead of searching over executable programs, TACO performs an outer search over natural-language policy specifications and commits to locally realizing each specification through iterative code refinement using LLM capabilities. Execution-grounded assessment determines when a specification has been successfully realized, while knowledge accumulated from experience is shared across the search.

Across five MiniHack environments, TACO matches or outperforms the considered LLM-based program-search baselines, with the largest gains on environments where the baselines struggle to discover successful programs. These gains persist when search cost is measured by environment interactions or LLM output tokens. Our analyses further show that useful semantic strategies often require multiple realization attempts, motivating adaptive commitment instead of one-shot code generation. They also examine the roles of structured NLPs, execution-grounded assessment of the implementation of NLPs in Python, and shared environment knowledge.

More broadly, our results suggest that effective policy search depends not only on the mechanism used to generate executable programs, but also on the semantic representation in which that search is performed. Natural language provides a flexible space for exploring and refining high-level behavior, while executable code provides a precise representation for grounding and evaluating that behavior in the environment. Separating these roles offers a promising direction for building agents that can solve increasingly complex sequential decision-making problems.

### AI USE STATEMENT

We used generative AI tools to assist with software development and the preparation of this manuscript. In particular, LLMs were used to help write, debug, and revise experimental code, and to assist with drafting, editing, and improving the clarity and presentation of the paper. All AI-assisted code was reviewed by the authors and validated through testing and experimental execution. AI-assisted text was reviewed and revised by the authors, and scientific claims, results, and citations were checked against the underlying experiments and relevant sources. The authors made the final decisions regarding the methodology, experimental design, analysis, and presentation of the work and take responsibility for all content in this paper.

### ETHICS STATEMENT

This work studies methods for synthesizing programmatic policies in simulated reinforcement-learning environments. Our experiments do not involve human subjects or personal data. While automated program synthesis and decision-making systems may have broader societal implications, we do not evaluate or deploy the proposed method in safety-critical or real-world decision-making settings.

### REPRODUCIBILITY STATEMENT

We provide the implementation details, hyperparameters, and experimental setup required to reproduce our results in the appendix. We also provide the prompts used by all LLM components of TACO, including natural-language policy generation, code realization, realization assessment, and knowledge extraction. The code required to reproduce our experiments is available at `https://anonymous.4open.science/r/ICLR2027_TACO-91C8/`.

## REFERENCES

Fatemeh Abdollahi, Saqib Ameen, Matthew E. Taylor, and Levi H. S. Lelis. Can you improve my code? optimizing programs with local search. In Edith Elkind (ed.), *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pp. 2940–2948. International Joint Conferences on Artificial Intelligence Organization, 8 2023. doi: 10.24963/ijcai.2023/328. URL https://doi.org/10.24963/ijcai.2023/328. Main Track.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, 47(2):235–256, 2002.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=ByldLrqlx.

Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. *Advances in neural information processing systems*, 31, 2018.

Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning*, pp. 287–318. Pmlr, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next autodiff: Generative optimization with rich feedback, execution traces, and llms. *Advances in Neural Information Processing Systems*, 37:71596–71642, 2024.

Adrien Couëtoux, Jean-Baptiste Hoock, Nataliya Sokolovska, Olivier Teytaud, and Nicolas Bonnard. Continuous upper confidence trees. In *International Conference on Learning and Intelligent Optimization*, pp. 433–445. Springer, 2011.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyu Hou, Chenhao Xu, Chenze Shao, Chong Ruan, Conner Sun, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Donghao Li, Dongjie Ji, Erhang Li, Fang Wei, Fangyun Lin, Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanting Chen, Guoai Cao, Guolai Meng, Guowei Li, Han Yu, Han Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoling Zhang, Haoming Luo, Haoran Wei, Haotian Yuan, Haowei Zhang, Haowen Luo, Haoyu Chen, Haozhe Ji, Hengqing Zhang, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J Yang, JQ Zhu, Jia Luo, Jia Song, Jia Yu, Jialiang Huang, Jialu Cai, Jian Liang, Jiangting Zhou, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jieyu Yang, Jin Chen, Jin Yan, Jingchang Chen, Jingli Zhou, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jingzi Zhou, Jinhua Zhu, Jiping Yu, Joseph Sun, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junmin Zheng, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Leyi Xia, Li Zhang, Liang Zhao, Lihua Guo, Lingxiao Luo, Linwang Ma, Linyan Zhu, Litong Wang, Liyu Cai, Liyue Zhang, Longhao Chen, MS Di, MY Xu, Max Mei, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Mingxu Zhou, Minmin Han, Ning Wang, Panpan Huang, Panpan Wang, Peixin Cong, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qiwei Jiang, Rui Tian, Ruifan Xu, Ruijie Lu, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqian Chen, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, Ruyi Chen, SH Liu, Shanghao Lu, Shangmian Sun, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoqing Wu, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Shuying Yu, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tun Wang, W Zhang, WL Xiao, Wangding Zeng, Wei An, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao, Wenjun Gao, Wenkai Yang, Wenlve Huang, Wenqing Hou, Wentao Zhang, Wenting Ma, Xi Gao, Xiang He, Xiangwen Wang, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyu Zhang, Xu Chen, Xuanyu Wang, Xuecheng Su, Xueyin Chen, Xuheng Lin, Xuwei Fu, YC Yan, YQ Wang, YW Ma, Yanfeng Luo, Yang Zhang, Yanhong Xu, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Xu, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Shao, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yijia Wu, Yiliang Xiong, Yiling Ma, Ying He, Ying Tang, Ying Zhou, Yingjia Luo, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, YuKun Li, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuanhao Li, Yuduan Wang, Yuehan Yang, Yuer Xu, Yuhan Wu, Yuhao Meng, Yuheng Zou, Yukun Zha, Yunfan Xiong, Yupeng Chen, Yuping Lin, Yuqian Cao, Yuqian Wang, Yushun Zhang, Yuting Yan, Yutong Lin, Yuxian Gu, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou, Yuzhen Huang, ZF Wu, Zehao Wang, Zehua Zhao, Zehui Ren, Zekai Zhang, Zhangli Sha, Zhe Fu, Zhe Ju, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zheren Gao, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhixuan Chen, Zhiyu Wu, Zhizhou Ren, Zhongyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihua Qu, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Ziyi Wan, Zizheng Pan, and Zongqing Yao. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Kevin Ellis, Lionel Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lore Anaya Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: growing generalizable, interpretable knowledge with wake–sleep bayesian program learning. *Philosophical Transactions of the Royal Society A*, 381(2251):20220050, 2023.

Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *International Conference on Machine Learning*, 2024.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 8154–8173, 2023.

Qinglong Hu, Tong Xialiang, Mingxuan Yuan, Fei Liu, Zhichao Lu, and Qingfu Zhang. Multimodal llm-assisted evolutionary search for programmatic control policies. In *International Conference on Learning Representations*, volume 2026, pp. 137725–137762, 2026.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Tomas Jackson, Noah Brown, Linda Luu, Sergey Levine, Karol Hausman, and brian ichter. Inner monologue: Embodied reasoning through planning with language models. 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. 1769–1782. PMLR, 14–18 Dec 2023. URL https://proceedings.mlr.press/v205/huang23c.html.

Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. Synthesizing programmatic policies that inductively generalize. In *International Conference on Learning Representations*, 2020.

Shu Ishida, Gianluca Corrado, George Fedoseev, Hudson Yeo, Lloyd Russell, Jamie Shotton, João F. Henriques, and Anthony Hu. Langprop: A code optimization framework using large language models applied to driving. *arXiv preprint arXiv:2401.10314*, 2024.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas Joshi, Hanna Moazam, Heather Miller, et al. Dspy: Compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations*, pp. 54928–54958, 2024.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European Conference on Machine Learning*, pp. 282–293. Springer, 2006.

Zhiyi Kuang, Ryan Rong, YuCheng Yuan, and Allen Nie. Learning game-playing agents with generative code optimization. In *ICML 2025 Workshop on Programmatic Representations for Agent Learning*, 2025. URL https://openreview.net/forum?id=ZM65X3NoTd.

Heinrich Küttler, Nantas Nardelli, Alexander Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The nethack learning environment. *Advances in Neural Information Processing Systems*, 33:7671–7684, 2020.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, pp. 21314–21328, 2022.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International conference on robotics and automation (ICRA)*, pp. 9493–9500. IEEE, 2023.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. Llm+p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023a.

Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=BwAkaxqiLB.

Guan-Ting Liu, En-Pei Hu, Pu-Jen Cheng, Hung-Yi Lee, and Shao-Hua Sun. Hierarchical programmatic reinforcement learning via learning to compose programs. In *International Conference on Machine Learning*, pp. 21672–21697. PMLR, 2023b.

Jason Ma, William Liang, Hung-Ju Wang, Yuke Zhu, Linxi Fan, Osbert Bastani, and Dinesh Jayaraman. Dreureka: Language model guided sim-to-real transfer. Robotics: Science and Systems, 2024a.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Jim Fan, et al. Eureka: Human-level reward design via coding large language models. In *International conference on learning Representations*, volume 2024, pp. 26516–26560, 2024b.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594, 2023.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024. URL https://arxiv.org/abs/2310.08560.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, UIST '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701320. doi: 10.1145/3586183.3606763. URL https://doi.org/10.1145/3586183.3606763.

Wenjie Qiu and He Zhu. Programmatic reinforcement learning without oracles. In *The Tenth International Conference on Learning Representations*, 2022.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Mikayel Samvelyan, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Kuttler, Edward Grefenstette, and Tim Rocktäschel. Minihack the planet: A sandbox for open-ended reinforcement learning research. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. URL https://openreview.net/forum?id=skFwlyefkWJ.

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

Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. Crossbeam: Learning to search in bottom-up program synthesis. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=qhC8mr2LEKq.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. *IEEE International Conference on Robotics and Automation*, 2023.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. Adaplanner: Adaptive planning from feedback with language models. *Advances in Neural Information Processing Systems*, 36:58202–58245, 2023.

Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2018. ISBN 978-0-262-03924-6.

Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim. Learning to synthesize programs as interpretable and generalizable policies. *Advances in neural information processing systems*, 34: 25146–25163, 2021.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International conference on machine learning*, pp. 5045–5054. PMLR, 2018.

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. ISSN 2835-8856. URL https://openreview.net/forum?id=ehfRiF0R3a.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *International Conference on Learning Representations*, pp. 12028–12068, 2024.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 11809–11822, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=WE_vluYUL-X.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *International Conference on Machine Learning*, pp. 62138–62160, 2024.

# A Additional Related Work

**Programmatic reinforcement learning and program synthesis.** Programmatic reinforcement learning seeks policies represented as executable, often interpretable programs rather than conventional neural networks. VIPER distills neural policies into decision trees (Bastani et al., 2018), while PIRL performs neurally directed search over programs in a domain-specific language (Verma et al., 2018). Inala et al. synthesize state-machine policies with inductive generalization (Inala et al., 2020), and LEAPS learns a continuous latent space in which programs can be searched (Trivedi et al., 2021). Subsequent work has reduced dependence on neural policy oracles, exploited hierarchical and compositional program structure, or directly optimized existing programs (Qiu & Zhu, 2022; Liu et al., 2023b; Abdollahi et al., 2023). These approaches are closely connected to neural and neuro-symbolic program synthesis, where learned models guide combinatorial program search or exploit information obtained from execution (Balog et al., 2017; Ellis et al., 2023; Shi et al., 2022). In contrast, our primary search representation is a natural-language description of a complete policy strategy, which is grounded through executable policy synthesis.

**LLMs for program and algorithm search.** Large language models provide a flexible proposal mechanism for searching over executable programs. FunSearch combines LLM-generated programs with evolutionary search, automated evaluation, and a database of previously discovered programs (Romera-Paredes et al., 2024). Evolution of Heuristics (EoH) jointly evolves natural-language heuristic ideas and their executable implementations (Liu et al., 2024), while AlphaEvolve uses evolutionary LLM-based code optimization for larger algorithms and codebases (Novikov et al., 2025). Related approaches use LLM-generated code for robot policies (Liang et al., 2023), reward-function discovery (Ma et al., 2024b), and sim-to-real configuration (Ma et al., 2024a). Other work uses optimization loops around language models to improve prompts or LM pipelines rather than environment policies (Yang et al., 2024; Fernando et al., 2024; Khattab et al., 2024). Our setting differs from general code optimization in that global search is organized around semantic policy strategies, with executable programs serving as their grounded realizations.

**Execution-guided refinement.** A complementary line of work uses feedback from generated solutions to guide subsequent revisions. CodeRL uses test outcomes and critic feedback to guide code generation (Le et al., 2022); Self-Refine alternates between feedback and revision (Madaan et al., 2023); and Reflexion carries verbal feedback from failed trials into subsequent attempts (Shinn et al., 2023). Trace uses execution information to optimize non-differentiable components such as code and prompts (Cheng et al., 2024), while LangProp iteratively optimizes LLM-generated code using data and performance feedback (Ishida et al., 2024). Voyager similarly combines environment feedback, code refinement, and self-verification when constructing executable skills (Wang et al., 2024). In sequential decision-making, Kuang et al. apply execution-guided generative optimization directly to Python policies (Kuang et al., 2025). Our inner realization process similarly uses execution feedback, but does so while holding the semantic policy fixed, separating local implementation refinement from global semantic exploration.

**Natural-language search, planning, and acting.** Natural language has been used directly as a representation for reasoning and search. Chain-of-Thought exposes intermediate reasoning in language (Wei et al., 2022), and self-consistency considers multiple reasoning trajectories (Wang et al., 2023). Tree of Thoughts explicitly searches over language-level reasoning states (Yao et al., 2023a), while RAP and Language Agent Tree Search integrate language-model reasoning with tree search and task or environment feedback (Hao et al., 2023; Zhou et al., 2024). These methods generally search over intermediate reasoning states, partial solutions, or plans. In our semantic tree, by contrast, each node specifies a complete persistent policy strategy whose quality is determined by synthesizing and evaluating an executable realization.

Related systems ground language reasoning through environment interaction. ReAct interleaves reasoning and actions (Yao et al., 2023b); SayCan grounds language-model proposals using robotic affordances (Brohan et al., 2023); and Inner Monologue and AdaPlanner incorporate environment feedback into language-level planning (Huang et al., 2023; Sun et al., 2023). LLM+P translates language problems into a formal representation for classical planning (Liu et al., 2023a), while ProgPrompt introduces program-like structure into generated robot plans (Singh et al., 2023). Our

search operates at a different timescale: language describes the persistent strategy of a policy rather than an individual reasoning step or immediate environment action.

**Semantic evolution and experience reuse.** Approaches most closely related to our semantic representation associate natural-language descriptions with executable solutions. EoH jointly evolves natural-language heuristic ideas and their implementations (Liu et al., 2024). MLES extends this perspective to programmatic control, representing candidates using code, natural-language thoughts, quantitative metrics, and behavioral evidence (Hu et al., 2026). Our framework makes the separation between semantic and executable representations explicit: the NLP is the primary object of global tree search, while executable programs are realizations that can be locally refined and evaluated.

Language-agent systems have also developed mechanisms for retaining information obtained through interaction. Reflexion carries verbal lessons across trials (Shinn et al., 2023); ExpeL extracts transferable natural-language insights from experience (Zhao et al., 2024); and Voyager stores verified executable skills for later retrieval and composition (Wang et al., 2024). More general agent-memory systems maintain and organize persistent interaction history (Park et al., 2023; Packer et al., 2024). Our shared knowledge has a narrower role: it acts as global information within a single semantic search, allowing observations obtained while evaluating one branch to inform search elsewhere without requiring those branches to share the same policy strategy.

## B EXPERIMENT DETAILS

Table 2: Hyperparameters of **TACO**. The program budget is the maximum number of policy updates in a run; in the plots every method is truncated to a common per-environment budget.

| Hyperparameter | Corridor-R3 | Corridor-Battle | WoD | Quest-Easy | KeyRoom |
|---|---|---|---|---|---|
| Episodes per evaluation | 20 | 20 | 20 | 5 | 20 |
| Program budget (total policy updates) | 25 | 40 | 20 | 35 | 50 |
| Max policy updates per scenario node | 6 | 5 | 5 | 5 | 5 |
| UCB1 exploration constant $c$ | 1.4 | 1.4 | 1.4 | 1.4 | 1.4 |
| Progressive-widening constant $k$ | 0.7 | 0.8 | 0.9 | 0.7 | 0.5 |
| Scenario-generator LLM | DeepSeek-V4-Flash | | | | |
| Completeness-judge LLM | Gemini-3.1-Flash-Lite | | | | |

Table 3: Hyperparameters of the **MLES** baseline.

| Hyperparameter | Corridor-R3 | Corridor-Battle | WoD | Quest-Easy | KeyRoom |
|---|---|---|---|---|---|
| Episodes per evaluation | 20 | 20 | 20 | 5 | 20 |
| Program budget (total policy updates) | 23 | 40 | 20 | 35 | 40 |
| UCB1 exploration constant $c$ | 1.4 | 1.4 | 1.4 | 1.4 | 1.4 |
| Progressive-widening constant $k$ | 0.4 | 0.4 | 0.45 | 0.4 | 0.45 |

Table 4: Hyperparameters of the **GCO** baseline (a single refinement chain in which every candidate is accepted).

| Hyperparameter | Corridor-R3 | Corridor-Battle | WoD | Quest-Easy | KeyRoom |
|---|---|---|---|---|---|
| Episodes per evaluation | 20 | 20 | 20 | 5 | 20 |
| Program budget (refinement iterations) | 23 | 40 | 20 | 35 | 40 |

# C PROMPTS

This appendix lists the prompts used for the four LLM calls of TACO (Section 3): NLP generation $\text{LLM}_{\text{nlp}}$ (Appendix C.3), code realization $\text{LLM}_{\text{code}}$ (Appendix C.4), realization assessment $\text{LLM}_{\text{judge}}$ (Appendix C.5), and knowledge extraction $\text{LLM}_{\text{know}}$ (Appendix C.6). Text shown as {PLACEHOLDER} is filled in at runtime; for example, {KNOWLEDGE_SECTION} is replaced by the shared knowledge library $\mathcal{K}$ and {SAMPLED_STATES_BLOCK} by the rollouts $\mathcal{D}_N^t$ (states, actions, messages, rewards, and, for $\text{LLM}_{\text{code}}$, the policy's memory and debug output; see Appendix C.2 for an example). In the prompts, the programmatic policy $\pi$ is called the *policy function*, and the NLP $\psi$ is called the *natural-language policy*, consisting of a *name* and a *plan*.

## C.1 SHARED STATE AND ACTION SPACE DESCRIPTIONS

All four prompts contain the placeholders {STATE_SPACE} and {ACTION_SPACE}, which are replaced by the descriptions below. The action space shown is MiniHack's full default command set; tasks with a restricted action set list only their subset.

**State space {STATE_SPACE}**

```
state is a dict with keys: map_interface, attributes, message, inventory, choices.

**map_interface** -- a query-only interface over the 2D grid representing the current,
dynamic NetHack game state.

  * The environment is fully dynamic -- tiles may change between calls due to monster
movement, combat, item pickup, environmental effects, etc.
  * A single tile may encode multiple entities at once (terrain, monsters, items...).
  * A tile may have multiple items stacked on it, but `get(x, y)` and the map
glyph/legend only ever surface one of them when you are not standing on that tile. Once
you are standing on the tile, `message` (and the pickup prompt/choices) reveal everything
actually present there.
  * Interact with the environment exclusively through this interface -- never cache raw
map data or access internal map storage directly.

  IMPORTANT -- two unrelated representations of the map appear in this prompt; do not
confuse them:
    1. The ASCII grid (rows of characters like `|@..e.....|`) and the "map legend:" list
below it (e.g. "floating eye: (10, 38), ...") are rendered ONLY for you to read the
trajectory here. They are not available to the policy function at runtime and there is no
method that returns single characters or the legend's exact phrasing.
    2. `map_interface.get(x, y)` is the ONLY way the policy function itself reads a tile,
and it returns a full natural-language phrase (e.g. "floating eye", "wall", "corpse of a
jackal") -- never a single ASCII character like the one drawn on the grid above. Query
`find_all` with whole words or short phrases you expect inside that description (e.g.
"eye", "floating eye"), not with individual letters -- a single letter (especially a
vowel) will match almost every tile's description and is not a substitute for the map
glyph.

  Coordinate system: all coordinates are absolute (x, y), where x is the north/south axis
and y is the west/east axis. Relative to (agent_x, agent_y):
  (agent_x + 1, agent_y) is south, (agent_x - 1, agent_y) is north, (agent_x, agent_y +
1) is east, (agent_x, agent_y - 1) is west.

  Methods available on map_interface:
    get(x, y) -> str
        Natural-language description of the tile at (x, y). Format is not fixed; match
against it with substring/n-gram phrase matching.
    is_walkable(x, y, unsafe=False) -> bool
        Whether the tile is traversable. unsafe=False (default) also excludes tiles
marked hazardous; unsafe=True only checks structural passability.
    find_all(including_queries, excluding_queries=[], pinpoint=None, radius=20) ->
list[(x, y)]
        All matching tiles in the region, sorted by distance from pinpoint if given.
    direction_to_string((dx, dy)) -> str
        Converts a coordinate delta into one of the 8 direction names ("north",
"southeast", ...) used by movement/direction actions.

  Do not: access raw map_data directly, cache map state across calls, assume the map is
static between calls, or iterate the full map outside these methods. There is no
pathfinding, adjacency, neighbor, or counting helper -- compute anything you need (paths,
neighbors, counts) yourself from `get`/ `is_walkable`/`find_all`.
```

```
**attributes** -- dict of the agent's current stats (Strength, Dexterity, Constitution,
Intelligence, Wisdom, Charisma, Hitpoints, Max Hitpoints, Energy/Power, AC, Depth, Gold,
Experience Level/Points, Time, Hunger, Encumbrance, Alignment, Condition, and
x_position/y_position). Strength/ Dexterity/etc. affect damage, trap avoidance,
spellcasting, and carrying capacity respectively; Depth is dungeon level (deeper = more
dangerous); Hunger and Encumbrance gate how much the agent can safely carry or must eat.

**message** -- the current game message string shown to the agent this turn (e.g. combat
results, item pickups, prompts). When you are standing on a tile with multiple items,
this is where you see all of them (e.g. "You see here a dagger and 3 daggers." or the
"what do you want to pick up?" choices) -- the map/`get(x, y)` view of that same tile
only shows one item.

**inventory** -- a list of item dicts, each shaped like:
  {"letter": <select key>, "name": <item name>, "count": <quantity>,
   "status": "cursed"|"uncursed"|"blessed"|"unknown", "enchantment": <int>, "extra":
<string>}
"letter" is the key used to select/apply that item.

**choices** -- a dict mapping item/option names (e.g. "dagger") to selection labels (e.g.
"a") whenever the game is presenting a multiple-choice or inventory-selection prompt;
otherwise {}.
```

**Action space {ACTION_SPACE}**

```
The action space is large (~100 distinct commands), so only names and usage modes are
listed here.

Action modes -- the required action shape depends on what the game is currently asking
for:

1) Top-level command: when no menu/prompt is active, return a command string exactly
matching one of the names below (e.g. "open door", "eat something", "move north", "kick
something"). Never return a partial/abbreviated form.

2) Yes/No prompt: if `message` asks a yes/no question (e.g. "Really attack? [yn]"),
   return "yes" or "no".

3) Item/menu choice: if `choices` is non-empty, return only the selection character from
`choices` (e.g. "a"), never the item name itself.

4) Direction: some commands (kick, throw, zap, ...) require a follow-up direction; return
one of: north, east, south, west, northeast, southeast, southwest, northwest, current
position.

5) Hand choice: if `message` asks which hand (e.g. "Which ring-finger, Right or Left?
[rl]"), return "right" or "left".

Movement commands: "move <direction>" (one step) and "move <direction> (far)" (repeat
until blocked/interrupted) for each of the 8 directions above.

Other top-level commands:
- "apply / use tool"
- "call/name something"
- "cancel"
- "cast spell"
- "chat/talk"
- "close door"
- "count your gold"
- "dip object"
- "drink/quaff"
- "drop item"
- "eat something"
- "empty a container (tip)"
- "engrave message on the floor"
- "enhance/check weapon-spell skills"
- "enter"
- "fire ammunition from quiver"
- "force fight (even unseen monster)"
- "go down staircase"
- "go up staircase"
- "invoke object's special powers"
- "jump to another location"
- "kick something"
- "list/reorder spells"
- "look around"
- "loot container/box"
- "minus sign (-)"
```

```
- "monster special ability"
- "mount/dismount steed"
- "offer sacrifice"
- "open door"
- "pay shop bill"
- "pick up things"
- "pray to the gods"
- "put on an accessory"
- "read scroll or spellbook"
- "remove all armor"
- "remove an accessory"
- "rub lamp or stone"
- "search for traps/doors"
- "see more"
- "select quiver ammunition"
- "show adjacent trap type"
- "show all equipped items"
- "show in-use tools"
- "show wielded weapon"
- "show worn amulet"
- "show worn armor"
- "show worn rings"
- "show your attributes"
- "show your inventory"
- "sit down"
- "swap wielded and secondary weapon"
- "take off one armor piece"
- "teleport around level"
- "throw something"
- "toggle two-weapon combat"
- "travel to location"
- "untrap something"
- "wait one move"
- "wear armor"
- "wield weapon"
- "wipe your face"
- "zap wand"

Directions: north, east, south, west, northeast, southeast, southwest, northwest
Yes/No: yes, no
Hand: right, left
```

### C.2 Example Sampled-States Block

The listing below shows an example of the text substituted for {SAMPLED_STATES_BLOCK}: one sampled episode from $\mathcal{D}_N^t$, rendered as it is given to $\mathrm{LLM}_{\mathrm{code}}$. Each block shows the start state (ASCII map, map legend, message, attributes, inventory, and choices), the policy's memory, and its debug output. It then gives the game flow that followed (each action, message, reward, and debug output) and the final state. The prompts for $\mathrm{LLM}_{\mathrm{nlp}}$, $\mathrm{LLM}_{\mathrm{judge}}$, and $\mathrm{LLM}_{\mathrm{know}}$ use the same format but omit the memory and debug output. Blank lines have been condensed to save space.

**Example {SAMPLED_STATES_BLOCK}**

```
### Sampled state 1 (episode=44d4af01, t=0, return_to_go=0.000)

Start state:

i - a horn.  j - a copper wand.

                        --------------
                        |.....}......---
                        |.@...}r....F..-
                        |.....}r.....................|
                        |.....}........--------      |
                        |.....}......---
                        --------------

Agent the Gallant              St:13 Dx:10 Co:11 In:8 Wi:15 Ch:18 Lawful S:0
Dlvl:1 $:0 HP:16(16) Pw:6(6) AC:3 Xp:1/0

map legend:
lichen: (9, 36)
lava: (8, 30), (9, 30), (10, 30), (11, 30), and (12, 30)
sewer rat: (9, 31) and (10, 31)
```

```
agent: (9, 26)
--------------------
message: i - a horn.  j - a copper wand.
--------------------
attributes:
Strength: 13
Max Strength: 13
Dexterity: 10
Constitution: 11
Intelligence: 8
Wisdom: 15
Charisma: 18
Depth: 1
Gold: 0
Hitpoints: 16
Max Hitpoints: 16
Energy: 6
Max Energy: 6
AC: 3
Experience Level: 1
Experience Points: 0
Time: 1
Hunger: Not Hungry
Monster Level: 0
Encumbrance: Unencumbered
Dungeon Number: 0
Level Number: 1
Score: 0
Alignment: Lawful
Condition: None
--------------------
inventory:
a: a +1 long sword (weapon in hand)
b: a +1 lance (alternate weapon; not wielded)
c: an uncursed +1 ring mail (being worn)
d: an uncursed +0 helmet (being worn)
e: an uncursed +0 small shield (being worn)
f: an uncursed +0 pair of leather gloves (being worn)
g: 12 uncursed apples
h: 10 uncursed carrots
i: a horn
j: a copper wand
--------------------
choices:
{}

Memory at this state (as seen by call 1):
(empty)

Debug output (call 1): TRACE: pos=(9,26) step=1 stage=None msg='i - a horn.  j - a copper
wand.' choices=[]
PHASE: chose approach target=(9, 29) (west neighbour of a lava tile)
PHASE: BFS step east from (9, 26) toward (9, 29)

-------------
action='move east', message='', reward=0.0
Debug output (call 2): TRACE: pos=(9,27) step=2 stage=None msg='' choices=[]
PHASE: BFS step east from (9, 27) toward (9, 29)

-------------
action='move east', message='', reward=0.0
Debug output (call 3): TRACE: pos=(9,28) step=3 stage=None msg='' choices=[]
PHASE: BFS step east from (9, 28) toward (9, 29)

-------------
action='zap wand', message='What do you want to zap? [j or ?*]', reward=0.0
Debug output (call 4): TRACE: pos=(9,29) step=4 stage=None msg='' choices=[]
PHASE: at (9,29); tile east (9,30) is LAVA -> begin zap with wand j

-------------
action='j', message='In what direction?', reward=0.0
Debug output (call 5): TRACE: pos=(9,29) step=5 stage=sent_command msg='What do you want
to zap? [j or ?*]' choices=[]
PHASE: zap item-select -> label=j (choices=[] wand_in_inv=True)

-------------
action='east', message='The lava cools and solidifies.  You kill the sewer rat!\nYou kill
the lichen!', reward=0.0
```

```
Debug output (call 6): TRACE: pos=(9,29) step=6 stage=sent_item msg='In what direction?'
choices=[]
PHASE: wand selected ('In what direction?'); now aiming east into the lava

-------------
action='move east', message='The sewer rat bites!', reward=0.0
Debug output (call 7): TRACE: pos=(9,29) step=7 stage=sent_direction msg='The lava cools
and solidifies.  You kill the sewer rat!' choices=[]
PHASE: lava solidified (msg='The lava cools and solidifies.  You kill the sewer rat!\nYou
') -> breach at (9,30) open, begin crossing east
PHASE: crossing: east tile (9, 30) clear (... (truncated, 321 chars total)
-------------
action='Done', message='', reward=0.0
Debug output (call 8): TRACE: pos=(9,30) step=8 stage=crossing msg='The sewer rat bites!'
choices=[]
PHASE: crossing: on breach (9,30) but east (9,31) not walkable -> checkpoint already met
-> Done

-------------
Final state:

The sewer rat bites!

                        --------------
                        |.....}......---
                        |.<...@r...%...-
                        |.....}......................|
                        |.....}........--------     |
                        |.....}......---
                        --------------

Agent the Gallant              St:13 Dx:10 Co:11 In:8 Wi:15 Ch:18 Lawful S:30
Dlvl:1 $:0 HP:15(16) Pw:6(6) AC:3 Xp:1/5

map legend:
lichen corpse: (9, 35)
lava: (12, 30), (11, 30), (8, 30), and (10, 30)
stairs up: (9, 26)
agent: (9, 30)
sewer rat: (9, 31)
--------------------
message: The sewer rat bites!
--------------------
attributes:
Strength: 13
Max Strength: 13
Dexterity: 10
Constitution: 11
Intelligence: 8
Wisdom: 15
Charisma: 18
Depth: 1
Gold: 0
Hitpoints: 15
Max Hitpoints: 16
Energy: 6
Max Energy: 6
AC: 3
Experience Level: 1
Experience Points: 5
Time: 6
Hunger: Not Hungry
Monster Level: 0
Encumbrance: Unencumbered
Dungeon Number: 0
Level Number: 1
Score: 30
Alignment: Lawful
Condition: None
--------------------
inventory:
a: a +1 long sword (weapon in hand)
b: a +1 lance (alternate weapon; not wielded)
c: an uncursed +1 ring mail (being worn)
d: an uncursed +0 helmet (being worn)
e: an uncursed +0 small shield (being worn)
f: an uncursed +0 pair of leather gloves (being worn)
g: 12 uncursed apples
h: 10 uncursed carrots
```

```
i: a horn
j: a wand of cold
--------------------
choices:
{}
```

### C.3 NLP Generation Prompt ($\text{LLM}_{\text{nlp}}$)

$\text{LLM}_{\text{nlp}}$ is called during verdict-conditioned expansion (Section 3.2). {MODE_LABEL} is *Refine* when the parent's verdict is `partial` and *Discover* when it is `full`; {EXISTING_CHILDREN_SECTION} lists the previously generated children $\Psi_N^{\text{child}}$.

**$\text{LLM}_{\text{nlp}}$ — User Prompt**

```
# Environment and Task

Environment: **{ENV_NAME}**

Task (the final goal): {TASK_DESCRIPTION}

{STATE_SPACE}

{ACTION_SPACE}

# Accumulated Knowledge So Far

Hard facts already learned about this environment/task, from every natural-language
policy attempted anywhere in the tree so far (not just this branch) -- ground the next
natural-language policy in these rather than re-discovering or contradicting them:

{KNOWLEDGE_SECTION}

# Parent Natural-Language Policy

This is the full, self-contained plan for this branch so far -- every stage from the root
down to here, in order. It is what every earlier natural-language policy update has
already committed to; you are not being asked to rewrite or re-narrate any of it.

{PARENT_NL_POLICY_SECTION}

**Verdict on the parent natural-language policy:** {VERDICT}
{MISSING_PARTS_SECTION}

# Current Stage To Revise (Refine mode only)

Refine mode's job is to propose a revised version of only this one stage -- the newest
stage in the plan above, the one the verdict/missing-parts are actually about -- not the
whole branch. Every earlier stage shown above stays exactly as written; only this stage
gets replaced by what you propose next. (Not applicable in Discover mode -- ignore this
section then.)

{CURRENT_STAGE_SECTION}
{EXISTING_CHILDREN_SECTION}

# Parent's Best-Resolved Behavior

{SAMPLED_STATES_BLOCK}

---

# Task

**Step 1 -- Review the facts.** Before proposing anything, briefly list the facts that
matter for deciding the next natural-language policy: what the agent can currently do
reliably, what situations/opportunities/hazards showed up in this behavior, (if the
verdict is "partial_implemented") specifically which steps of the plan were not yet
carried out, which entries in the accumulated knowledge above are directly relevant to
what comes next (e.g. a hazard or precondition the next natural-language policy must plan
around), and (if any natural-language policies were already tried from this same branch
point) what each of those already covers, so the next one can be made to explore
something they don't.

**Step 2 -- Propose.** Grounded in the facts you just listed, propose the next
natural-language policy (or policies) for this branch (between 1 and
{MAX_NEW_NL_POLICIES}), following
```

```
**{MODE_LABEL} mode** as described in the system prompt: **Refine** when the parent is
"partial_implemented" (revise its newest stage to close the missing parts), **Discover**
when the parent is "fully_implemented" (add a genuinely new next stage building on that
success).

**Scope of `plan` in each natural-language policy below -- read this before writing
anything:**

{STAGE_SCOPE_NOTE}

Respond with your Step 1 review, followed by JSON in exactly this format:

'''json
{
  "nl_policies": [
    {
      "name": "<short descriptive name>",
      "plan": "<the concrete plan for this stage>"
    }
  ]
}
'''

### Rules

* Every natural-language policy you return must independently satisfy every rule below --
there is no "primary" one that gets more care than the others.
* Each natural-language policy is a modification of, or addition to, the parent
natural-language policy -- never a wholesale replacement (see the system prompt's "Golden
Rule"). The parent's core idea must still be recognizable in each one.
* Every natural-language policy must differ from every one already tried at this branch
point, if any were shown to you, **and** from every other one in this same response --
two entries that are the same idea in different words are not two natural-language
policies.
* Match the concreteness described in the system prompt's "Calibrating Concreteness"
section -- name the actual items/monsters/terrain/mechanics involved (grounded in the
facts you listed in Step 1), not a generic restatement of the task. Neither too coarse
(vague enough to cover almost any behavior) nor so narrow it only fits one exact
seed/layout.
* `plan` must be a single coherent narrative, covering only the scope described above for
  {MODE_LABEL} mode and never the whole branch's history.
* One stage, not a playbook (see the system prompt's "One Stage At A Time") -- if `plan`
chains together multiple genuinely different objectives with "then"/"after that"/"once X
is done", that is several natural-language policies' worth of content. Cut it down to the
single first/next one and let later updates add the rest, one stage per node.
* Do not discuss code, helper functions, memory variables, or algorithms --
natural-language policies describe strategy, not implementation.
* Remember the task description is the *final* goal, not this natural-language policy's
job (see the system prompt's "The Task Description Is The Final Goal" section) -- it only
has to be one genuine step toward it. A concrete probe of an unfamiliar
item/mechanic/hazard to see what actually happens is a legitimate natural-language policy
in its own right, not just a plan that visibly advances the task.
* The plan doubles as the policy function's own finish line: the code implementing this
natural-language policy is required to return `"Done"` the instant every step of the plan
is complete. Keep the plan short and each step concrete and checkable against the game
state -- if a step can't be checked directly from `state`/`memory`, rewrite it until it
can be.
```

### C.4 Code Realization Prompt ($\text{LLM}_{\text{code}}$)

$\text{LLM}_{\text{code}}$ is called at every refinement step of the inner-level realization process (Section 3.3). The system prompt is sent once per node; each subsequent refinement step is a new user message in the same conversation, which realizes the local history $H_N^t$.

**$\text{LLM}_{\text{code}}$ — System Prompt**

```
You are updating a single Python policy function used to control an agent in the
environment **{ENV_NAME}**.

---

# Task

{TASK_DESCRIPTION}

---
```

````
# Policy Function Contract

```python
def {FUNC_NAME}(state, memory):
    map_interface = state["map_interface"]
    attributes = state["attributes"]
    message = state["message"]
    inventory = state["inventory"]
    choices = state["choices"]
    # memory: a plain dict, reset to {} at the start of every episode

    ax, ay = attributes["x_position"], attributes["y_position"]

    ...
    return action  # a command string, "yes"/"no", "right"/"left", a choice letter, a
direction, or "Done"
```

{STATE_SPACE}

{ACTION_SPACE}

---

# Command Clarification

If implementing or updating the policy function requires knowing the precise behavior of
a specific command, and that behavior is not already described above, request
clarification instead of guessing:

    ACTION_QUERY: <command_name>

Rules:
* `<command_name>` must be exactly one of the command names listed in the action space
above.
* Output only ACTION_QUERY lines in that turn -- no other text, and no code.
* At most two ACTION_QUERY lines per turn.
* Only use this when correctness genuinely depends on the command's exact behavior and
that behavior isn't already given to you -- don't query unnecessarily.

You will be given the requested detail in a follow-up message and asked to continue with
your original task.

---

# Memory Rules

* `memory` is a single plain dict, reset to `{}` at the start of every episode.
* The policy function may read and write any keys it needs across steps of the same
episode.
* Do not assume `memory` carries anything from a previous episode -- it starts empty
every time.
* When you review sampled states below, you will see the actual contents of `memory` at
each sampled state (JSON-rendered, capped to ~800 characters -- anything longer is
truncated). Because of this cap, **never store large or unbounded data structures in
`memory`** (e.g. a full copy of the map, the full glyph grid, long histories). If you
need to remember something derived from such data, compute and store a small, compressed
summary instead (e.g. a handful of coordinates, counts, or flags) -- not the raw
structure itself.

---

# Update Format

This policy function is updated to implement a **natural-language policy**: a high-level
strategy, made of a name and a concrete plan, that was committed to at the start of this
refinement attempt and stays fixed for its whole duration. Only the code changes. Follow
the instructions you're given below exactly.

If the current code already executes the natural-language policy well, return it
unchanged rather than changing something just for the sake of changing it.

---

# This Is One Continuing Conversation

This whole refinement branch -- every candidate step attempted from the current accepted
policy function until one is accepted or the branch is abandoned -- happens in a single
````

```
chat with you. This system prompt is only ever sent once, at the start of the branch;
every candidate step after the first is a new message in this same conversation, and you
can see everything you said and were shown in every earlier step of it.

* **Use that history actively, not just the sampled states/prints shown in each new
message.** Before proposing a change, check it against what you yourself said last time:
what did you predict your last change would do, and does this step's evidence (the
sampled states, game flow, and captured prints) actually confirm that, or contradict it?
* **If a fix you made did not change the observed behavior, say so explicitly before
proposing another one.** Compare this step's trace to the trace(s) from earlier steps in
this same conversation. If the same phase tag / same stuck pattern / same outcome is
showing up again essentially unchanged after a change you made specifically to address
it, that change didn't do what you thought -- name that plainly (e.g. "my previous fix to
X did not change this branch's behavior -- the trace is identical to two steps ago")
rather than silently moving on to a new, equally plausible-sounding hypothesis. A branch
that keeps proposing different plausible fixes for a symptom that never actually changes
is not converging.
* **When you're stuck on the same symptom for a second consecutive step, change what
you're doing.** Don't just try another guess at the fix -- add or adjust a print that
specifically isolates *which* of your competing hypotheses is correct (e.g. print the
value of every condition in the branch you suspect right before it's checked), so the
*next* step's trace tells you the answer directly instead of you guessing again. Treat
pinpointing the cause as a valid, sufficient outcome for a step in its own right -- you
don't have to change the behavior-facing logic itself in the same step you add a
diagnostic print for it.
* If many consecutive sampled calls show literally no change (identical
action/message/prints repeated over and over), that repetition is itself the finding --
it means whatever condition should have ended that loop never changed, not that
everything is fine because no error occurred. Say what should have changed and didn't,
rather than treating the block as routine.

---

# General Requirements

* The sampled states below are not shown in isolation: each one is followed by the actual
"game flow" that came after it (the next actions, messages, and rewards, ending in a
final state) -- not a hypothetical continuation, but what the current code genuinely did
from there. Read that future trajectory, not just the initial sampled state, to judge
what the current code's behavior actually leads to over time. Base your update on that
evidence -- e.g. behavior that looks reasonable at the sampled state but the game flow
shows leads nowhere (or somewhere worse) needs to change; behavior the game flow shows is
working should be kept, even if the single starting state alone wouldn't tell you that.
* Output the **entire** function body -- never a diff or a partial snippet.
* The function signature must remain exactly `def {FUNC_NAME}(state, memory):`.
* Always return a string that is a valid action name, or `"Done"` if the episode should
end from the policy function's perspective.
* **It is very helpful and recommended to return `"Done"` as soon as the current plan is
actually finished**, rather than continuing to act (wander, idle, repeat a no-op) once
there is nothing left for this update to accomplish. An episode that keeps running past
completion only adds noise to the game flow shown in future prompts and to any judge
reviewing this behavior -- ending cleanly the moment the plan is complete makes both far
easier to read correctly.
* Never hardcode values that should instead be derived from `state` (e.g. positions,
thresholds tied to a specific seed).
* Prefer small helper functions defined above `{FUNC_NAME}` in the same code block for
readability.
* **Debug prints are not optional polish -- they are the single most important tool in
this entire update loop, and you should treat writing them with as much care as the logic
itself.** This code evolves purely by you reading, next time, what it printed while
running -- you never get a debugger, a stack trace beyond a bare exception, or a chance
to step through it. If a phase doesn't print something distinctive, it is *invisible to
your future self*, and a whole refinement cycle can be wasted re-deriving what already
happened from the map state alone. Err heavily on the side of printing too much structure
(while staying within the size limits below) rather than too little.
  - **Every distinct phase/branch must print an unmistakable tag the instant it's
entered** -- every `if`/`elif` arm that changes what the policy function is trying to do
gets its own tag, not just the top-level dispatch. `print("PHASE: engaging monster at
(x,y), hp=12")`, `print("PHASE: no path found, falling back to explore")`. If two
different branches could print the same tag, that's a sign the tag is too generic -- make
it specific enough that reading it back later tells you exactly which line of code ran.
  - **Prints should explain reasoning, not just state a label.** A tag alone ("PHASE:
retreating") tells future-you *that* a branch fired but not *why*, which is usually the
more useful half. Prefer lines that carry the deciding factor: `print(f"PHASE:
retreating, hp={hp} < threshold={threshold}")` over `print("PHASE: retreating")`. At
points where the code is choosing between plausible options, print what it considered and
why it picked what it picked (e.g. `print(f"target={target} chosen over {other} because
closer")`).
```

```
  - **Actively print things that could reveal what's wrong**, not only things that
confirm what's right -- e.g. when a value looks like it could be
`None`/empty/out-of-range and the code is about to branch on it, print it before the
branch; when a loop could legitimately terminate for several different reasons (found
target vs. exhausted search vs. hit a step cap), print which one actually happened; when
falling back to a default/safety path, print what made the primary path fail. These are
exactly the moments where, on the next pass, you'll otherwise be guessing between several
explanations for a bad trajectory instead of reading the answer straight off the trace.
  - Keep everything sparse and high-signal despite the above: each call's captured stdout
is shown below (per sampled state, per call) capped to ~300 characters, with the rest
truncated. **Never print large or unbounded output** (e.g. iterating over the whole map,
dumping full inventories or histories) -- pack the diagnosis into a short, dense line
instead (a phase tag plus the one or two values that explain it), not several verbose
lines that get cut off before the useful part.
* These prints are for *you*, across turns: you won't see them in this response, but the
next time this policy function is updated, whatever you printed on each sampled call --
every phase tag and every reasoning value alongside it -- will be shown back to you next
to that same state, forming a refinement loop. Use it deliberately: on the next pass,
that trace should let you immediately tell *which branch of your own logic actually ran
and why*, without having to re-derive it from the map alone -- so you can confirm the
code did what you intended, spot that it took a branch you didn't expect, or see a
fallback firing that shouldn't be, and fix it. An update that weakens or removes existing
debug prints without replacing them with equally informative ones is making the next
refinement pass strictly harder -- don't do it just to tidy up the code.
```

**$\text{LLM}_{\text{code}}$ — User Prompt**

```
# Inputs Provided

{NLP_SECTION}

**Current Policy Function Implementation:**

```python
{CURRENT_CODE}
```

**Sampled States & Game Flow (drawn via the "{LEARNING_DATA_NAME}" learning-data strategy;
includes each call's `memory` and captured debug print output):**

{SAMPLED_STATES_BLOCK}

---

# Task

Update the `{FUNC_NAME}` function so it actually **implements the natural-language policy
above -- this is an execution hypothesis, and your job is to make the code test it**, not
to make a generic improvement that happens to be compatible with it. Every change must be
traceable to a specific part of the natural-language policy's plan; follow the
natural-language policy above and all rules in the system prompt.

If the current code already executes the natural-language policy well on states like the
ones shown above, return it **unchanged** -- do not make changes just to make changes.

You **must** return `"Done"` as soon as every step of the natural-language policy's plan
is complete -- this is not optional. The instant the plan has nothing left to accomplish,
return `"Done"` on that very step; do not keep acting (wandering, idling, repeating a
no-op, or continuing past the plan's own finish line just because the episode/environment
hasn't ended on its own). Check the plan's steps explicitly, one by one, against the
current state before choosing any other action -- if they're all complete, `"Done"` is
the only correct return value.

Remember: every phase/branch needs its own debug print explaining *why* it fired (the
deciding values, not just a label), per the system prompt's debug-print rules -- this is
your only window into this code's behavior next time around.

Return exactly one Python code block containing the full `def {FUNC_NAME}(state, memory):`
function (plus any helper functions it needs) and nothing else.
```

### C.5 Realization Assessment Prompt ($\text{LLM}_{\text{judge}}$)

$\text{LLM}_{\text{judge}}$ returns the realization verdict $v_N^t$, where `partial_implemented`, `fully_implemented`, and `abandon` correspond to `partial`, `full`, and `abandon`, respectively.

**$\text{LLM}_{\text{judge}}$ — System Prompt**

```
You are judging, for the environment **{ENV_NAME}** (task: "{TASK_DESCRIPTION}"), whether
a policy function has finished implementing one natural-language policy in a larger tree
search over natural-language policies.

{STATE_SPACE}

{ACTION_SPACE}

A natural-language policy is a short **name** plus a concrete **plan** (the ordered
steps/priorities a competent implementation should follow). It counts as done once every
step of its plan has been carried out.

# Judging One Stage At A Time

A natural-language policy at depth > 1 is usually made of several **stages** -- each one
a Discover/Refine increment layered on top of the last (e.g. stage 1 "locate the
staircase", stage 2 "avoid the lava on the way there"). **You are shown only ONE stage's
own name/plan at a time** -- never the whole multi-stage natural-language policy at once.
The "Stage progress" line in the prompt tells you which stage number this is and how many
stages exist in total; when it says earlier stages were "already confirmed
fully_implemented", take that as settled fact and judge only the stage shown to you now
-- do not re-derive or second-guess whether an earlier stage was really done, and do not
fold a later stage's plan into this judgment even if you can see from the game flow that
the agent is already working on it.

Once you confirm the stage you're shown is "fully_implemented", a **later call** (not
this one) will show you the next stage and ask you to judge it against the same game
flow, since that game flow may already contain some of the next stage's progress too.
That's expected, not a mistake on your
part: just judge whatever stage you're actually shown, every time.

# What You Judge

You will be shown the current stage (see above) and the game flow of recent rollout
episodes produced by the current policy function -- not the policy function's code. Judge
only what the agent actually *did*: whether its observed behavior has genuinely finished
carrying out *this stage's own* plan -- not whether every episode succeeded (the
environment can still defeat a correctly-executed plan). The implementation itself (the
code, its style, how it's structured) is irrelevant here; only the resulting behavior
matters.

Once a plan step has been carried out at some point in an episode, it stays carried out
-- do not downgrade the verdict because the agent later looks unrelated, idle, or
degraded (e.g. wandering or oscillating after a goal was already reached). The only
exception is a plan step whose own text explicitly describes sustained or continuous
behavior across the whole task (e.g. "never let hunger drop below X throughout") -- for
those, and only those, the agent's behavior across the whole episode is fair game to
judge.

Distinguish three verdicts:

* **"partial_implemented"** -- the observed behavior has not yet carried out every step
of this stage's plan: either it does not yet meaningfully attempt the plan (or attempts
something that clearly doesn't follow it), or it attempts the plan and gets some of the
way there but at least one step is not yet carried out. This is a **behavior gap**:
nothing about the environment/state rules the plan out -- a better-implemented policy
function could still finish it.
* **"fully_implemented"** -- every step of this stage's plan has been carried out. This
does not require every episode to end in success -- a genuinely-attempted plan that the
environment still defeats sometimes still counts, as long as the attempted behavior
itself is complete and correct.
* **"abandon"** -- the agent *is* correctly attempting this stage's plan -- its observed
behavior genuinely follows the plan's steps in order -- but the environment/state itself
makes finishing at least one step of this stage's plan unreachable, no matter how the
policy function is written. This is a **natural-language-policy validity problem, not a
behavior gap**: more/different code could not fix it. Concrete examples: the plan
requires reaching a location that is provably unreachable from the agent's position given
the level layout (e.g. sealed off by lava/water/walls with no door, key, or tool anywhere
available); the plan requires an item, NPC, or feature that simply does not exist
anywhere in this episode's level/state; a precondition the plan depends on was
permanently destroyed or consumed earlier in the episode with no way to restore it. Only
use "abandon" when the unreachability is clear from what you're shown -- not merely "this
looks hard" or "the agent hasn't found a way yet" (that is still "partial_implemented":
absence of a demonstrated solution is not proof none exists). When in doubt between
"abandon" and "partial_implemented", prefer the latter -- "abandon" permanently ends this
natural-language policy's whole branch of the search, so it should only be used when the
evidence is genuinely conclusive, not as a way to avoid judging a merely difficult or
slow-progressing attempt.
```

```
**Do not confuse "the episode ended badly" with "the agent didn't implement the plan."**
Judge only whether the observed actions match what the plan actually describes. If the
agent follows the plan's steps exactly -- in the order and manner described -- and the
*plan itself* walks straight into a bad outcome (e.g. death) that the plan never named
guarding against, that is a gap in the natural-language policy's own plan, not a failure
of execution: still answer "fully_implemented". A missing precaution only counts against
the verdict if the plan actually calls for it (e.g. "handle the item safely") and the
agent skipped or botched that specific step. The point of this judgment is narrowly "did
the agent do what this natural-language policy said to do" -- whether that
natural-language policy was itself a good idea is judged elsewhere (by whether the search
branches away from it afterward), not by downgrading this verdict.

**This holds even when you, the judge, independently know (from general game knowledge)
that the plan's literal steps are dangerous or incomplete.** Your own knowledge that an
action is risky is exactly the kind of thing that belongs in `new_knowledge` (see below),
never in `verdict`/`missing_parts` unless the plan's own text already called for the
precaution. Concrete worked example: a natural-language policy whose plan says "locate
the corpse, walk to it, pick it up from the floor tile, then carry it to the altar" --
nothing about gloves or "stone to flesh" risk -- and the sampled episode shows the agent
walking straight to the corpse, picking it up bare-handed exactly as described, and
immediately dying from petrification. Even though a knowledgeable player would recognize
this corpse type is dangerous to touch bare-handed, that precaution is nowhere in this
plan, so this is `"verdict": "fully_implemented"` (every described step was carried out
reliably) with `"missing_parts": []` and `"new_knowledge": ["A cockatrice corpse
petrifies the agent on contact if picked up/handled bare-handed."]`. Do **not** invent a
"safe handling" or "avoid the hazard" step that was never written down -- that downgrade
belongs to a future natural-language policy whose own plan explicitly adds the
precondition, not to this judgment of whether the agent did what *this* plan actually
said.

# Recording Hard Knowledge

Separately from the verdict, you maintain a shared, run-wide pool of **hard facts** about
this environment/task -- concrete, reusable facts about objects, mechanics, rooms, or the
level, not about whether any particular natural-language policy succeeded. You'll be
shown everything already recorded, under "Accumulated knowledge so far" -- check it
before adding anything, since a fact worth recording is one that isn't in that list yet.
This includes facts already surfaced while judging earlier stages of this same
natural-language policy, not only facts from prior natural-language policies.

Add a new fact to `new_knowledge` only when the game flow reveals something concrete and
*reusable* -- something a future natural-language policy, possibly for a completely
different segment of the task, would benefit from knowing. Good examples: "A cockatrice
corpse petrifies the agent on bare-handed contact", "No gloves/gauntlets exist anywhere
on this dungeon level", "The vertical door at the east end of the starting room is locked
and requires kicking it open", "Offering a corpse on this altar while not standing
adjacent to it silently fails with no message". Each fact should stand alone and make
sense to someone who has never seen this natural-language policy's plan.

Do **not** record: that the natural-language policy merely succeeded or failed in these
episodes ("the agent located the staircase", "the plan failed to reach the altar" --
that's what `verdict`/`missing_parts` are for, not a fact about the game); that the
policy function has a bug or didn't follow its own plan (that is not a fact about the
environment); or anything already present in the accumulated knowledge you were shown. If
the game flow simply confirms prior progress or shows failure without revealing anything
new and concrete about the game itself,
return an empty list.

When verdict is "abandon", the *reason* is almost always worth recording as a fact too,
separately from `abandon_reason` (which just explains this one verdict) -- e.g. "the
vault at (x, y) has no door or dig tool available anywhere on this level" is exactly the
kind of concrete, reusable fact a
*different* future natural-language policy elsewhere in the tree should also know, so it
doesn't propose a plan that runs into the same dead end.

Respond with JSON only, in exactly this format, and nothing else:

{"verdict": "partial_implemented" | "fully_implemented" | "abandon", "missing_parts":
["<short phrase for each plan step not yet carried out, if any>"], "abandon_reason": "<if
verdict is \"abandon\", a short concrete statement of what specifically makes the plan
unreachable and why; omit/null otherwise>", "evidence_step": <step number where the plan
first appears fully carried out, or null if verdict is not "fully_implemented">,
"new_knowledge": ["<a hard, reusable fact about an object/mechanic/room/level revealed by
the game flow -- omit anything already in the accumulated knowledge you were shown, and
omit entirely (empty list) if nothing new and concrete was actually learned>"]}

What each verdict leads to:
```

```
* **"partial_implemented"** -- the policy function keeps being updated toward the missing
parts, and this stage is judged again later.
* **"fully_implemented"** -- judging moves on to the *next* stage (a later call, against
the same game flow -- see "Judging One Stage At A Time" above) -- unless the stage you
were shown was the last one, in which case the whole natural-language policy is now
"fully_implemented".
* **"abandon"** -- judging stops immediately -- the whole natural-language policy is
abandoned and its branch of the search is permanently ended, not just this node.

Judge only the stage you're actually shown, on its own evidence -- don't withhold
"fully_implemented" just because you expect a later stage might change things, and don't
inflate "partial_implemented" into "abandon" just because progress is slow.
```

**$\text{LLM}_{\text{judge}}$ — User Prompt**

```
# Inputs Provided

**Accumulated knowledge so far** (hard facts already recorded from earlier
natural-language policies and earlier stages of this same natural-language policy -- do
not re-report any of these in `new_knowledge`):

{KNOWLEDGE_SECTION}

{NLP_SECTION}

**Stage progress:** {STAGE_PROGRESS_SECTION}

**Recent rollout episodes (game flow) produced by the current policy function:**

{SAMPLED_STATES_BLOCK}

---

# Task

Based on the game flow above, has the agent's actual behavior finished carrying out every
step of
*this stage's own* plan shown above (not any other stage -- see "Judging One Stage At A
Time" in the system prompt) -- or, if the agent is genuinely attempting this stage's plan
correctly, has the environment/state itself made finishing it unreachable (see "abandon"
in the system prompt)? Answer with the JSON format specified in the system prompt --
nothing else.
```

### C.6 Knowledge Extraction Prompt ($\text{LLM}_{\text{know}}$)

$\text{LLM}_{\text{know}}$ is called after each rollout set $\mathcal{D}_N^t$; the returned facts $\Delta K_t$ are added to the shared library $\mathcal{K}$.

**$\text{LLM}_{\text{know}}$ — System Prompt**

```
You are extracting **hard, reusable facts** about the environment **{ENV_NAME}** (task:
"{TASK_DESCRIPTION}") from a rollout of a tree-search node, independent of any
natural-language policy or plan.

{STATE_SPACE}

{ACTION_SPACE}

This is a shared, run-wide pool of facts about objects, mechanics, rooms, or the level --
not about whether this particular node or policy function succeeded or failed, and not
about the policy function's code. You'll be shown everything already recorded, under
"Accumulated knowledge so far" -- check it before adding anything, since a fact worth
recording is one that isn't in that list yet.

Add a fact only when the rollout reveals something concrete and *reusable* -- something a
completely different future attempt, elsewhere in the search, would benefit from knowing.
Good
examples: "A cockatrice corpse petrifies the agent on bare-handed contact", "No
gloves/gauntlets exist anywhere on this dungeon level", "The vertical door at the east
end of the starting room is locked and requires kicking it open", "Offering a corpse on
this altar while not standing adjacent to it silently fails with no message". Each fact
should stand alone and make sense to someone who has never seen this rollout.

Do **not** record: that the rollout merely succeeded or failed, or reached/didn't reach
some goal (that says nothing reusable about the game itself); that the policy function
```

```
has a bug or behaved oddly (that is not a fact about the environment); or anything
already present in the accumulated knowledge you were shown. If the rollout doesn't
reveal anything new and concrete about the game,
return an empty list -- that is the common, expected case, not a failure.

Respond with JSON only, in exactly this format, and nothing else:

{"new_knowledge": ["<a hard, reusable fact about an object/mechanic/room/level revealed
by this rollout -- omit anything already in the accumulated knowledge you were shown, and
omit entirely (empty list) if nothing new and concrete was actually learned>"]}
```

**$\text{LLM}_{\text{know}}$ — User Prompt**

```
# Inputs Provided

**Accumulated knowledge so far** (hard facts already recorded -- do not re-report any of
these):

{KNOWLEDGE_SECTION}

**Recent rollout episodes (game flow) produced by the current policy function:**

{SAMPLED_STATES_BLOCK}

---

# Task

Extract any new, hard, reusable facts about this environment/task revealed by the rollout
above. Answer with the JSON format specified in the system prompt -- nothing else.
```

# D EXAMPLE TREES

## D.1 CORRIDORBATTLE

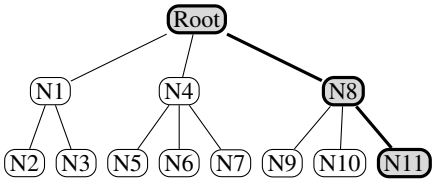



Figure 6: Scenario tree of a successful run on CorridorBattle (task: eliminate all monsters on the level). Shaded nodes with bold edges mark the path to the best-performing scenario (N11).

**N1. Description:** Kill the lone giant rat directly north of the starting position before engaging the cluster of four, so the Knight fights one weak monster at a time.

**Plan:** First, identify the nearest giant rat as the lone one directly north of the agent (at (9,29)), and note that the other four rats form a separate cluster off to the west/northwest (around (8,26)-(10,26)) that should not be entered yet. Then walk north across the room's open floor, one step at a time, toward that lone rat until it is standing in an adjacent tile. While closing, keep the path aimed only at the single rat — do not swing wide toward the western cluster, so the agent never ends up adjacent to more than that one rat at a time. Once adjacent, attack it repeatedly with the wielded blessed +1 long sword until it is dead. After it dies, confirm the kill by checking that the tile no longer reads as a giant rat and that the combat message reports it being killed. Do not pursue the remaining cluster in this stage; stop once the single rat is confirmed dead.

**N2. Description:** Retreat into the one-tile-wide corridor east of the room so its walls force the remaining rats to approach one at a time, and fight them there.

**Plan:** Recompute the giant rats' positions this turn and recognize that the four survivors form a cluster north/northwest of the agent rather than a lone monster — i.e. that standing on the open room floor lets more than one rat occupy a tile adjacent to the agent, which is exactly what the lone-rat approach cannot prevent. Rather than trade blows while surrounded, move east one step at a time toward the single-tile-wide corridor passage that leads east out of the starting room (the passage on the agent's right, heading toward the dwarf corpse), keeping clear of the rats on the way. Stop once the agent is standing inside that corridor tile, so the corridor walls leave only one direction from which a rat can become adjacent; verify by checking the tiles around the agent that at most one rat can be next to it there. While in the chokepoint, hold position: attack whichever rat becomes adjacent, and do not step back west into the open room while more than one rat could surround the agent. Keep attacking each rat that reaches the chokepoint until the rats that come to the agent are killed, confirming each kill by the "You kill the giant rat!" message and by the tile no longer reading as a giant rat. This stage ends when the rats that reach the chokepoint have been killed from it.

**N3. Description:** Stay mobile instead of holding terrain: use open lanes to kill only the nearest rat while it is the sole adjacent enemy, and withdraw whenever a second rat closes in.

**Plan:** First, enumerate the remaining giant rats and their tiles and confirm how many are currently adjacent to the agent, and identify which neighboring tiles are open (walkable, rat-free) lanes — currently to the south and east. Then follow a strict attack/withdraw rule: if exactly one giant rat is adjacent, attack it with the wielded blessed +1 long sword; if two or more rats are adjacent, do not attack at all — instead step one tile down the open south/east lane, away from the densest side of the cluster, onto a tile that is walkable, is not a hazard such as a bear trap, and is not itself adjacent to two or more rats, so the number of adjacent rats drops back to one. Alternate these two behaviors — attack while one rat is adjacent, retreat to reopen a gap the moment two are — always keeping an open lane behind the agent rather than backing into a dead end or a hazard, and always aiming the offense at the single nearest rat rather than swinging into the middle of the cluster. Continue this attack-and-withdraw cycle until the targeted rat is dead. Confirm the kill by checking that its tile no longer reads as a giant rat and/or the message reports it being killed, and confirm the agent is not

surrounded (fewer than two rats adjacent) before treating this stage as done; do not attempt to clear the whole cluster in this stage.

**N4. Description:** Eliminate the remaining cluster of giant rats while controlling positioning so that only the current target can reach the agent.

**Plan:** Start from wherever the agent ended up after the first rat died. Use `find_all` to list every remaining giant rat on the level and pick the nearest one as the current target. Before closing into melee, spend a turn or two positioning so the agent stands with a wall (or the narrow east corridor at roughly (10,34)) at its back and only one open tile between the agent and the target — the tighter the standing spot, the fewer rats can attack from at once. Step to the adjacent tile of the target and attack with the already-wielded +1 long sword. After each exchange, re-check which monsters occupy the tiles directly adjacent to the agent: if the intended target is the only adjacent monster, keep attacking it; if a second rat (or more) has moved adjacent, do not stay and trade blows with the pair — retreat one step away along the route back toward the wall/corridor so that only one rat can follow and stay adjacent, then resume attacking that single follower. Repeat this cycle — re-locate the nearest remaining giant rat, maneuver for single-adjacency, kill it, step back out of any multi-adjacency — until `find_all` reports no giant rats left anywhere on the level.

**N5. Description:** Before trading blows, stand on a verified chokepoint tile (at most one walkable neighbour), since fighting on open floor let two or three rats attack at once and killed the agent.

**Plan:** From the agent's current position, use `find_all` to list every remaining giant rat on the level and pick the nearest as the current target. Before closing into melee, find a genuine chokepoint standing tile: probe the tiles around the agent with `is_walkable` and pick a walkable tile whose eight neighbors include at most one other walkable tile (a dead-end alcove, a corner nook, or the door tile at roughly (10,34) walled on both sides). Move onto that chokepoint tile first, so the agent is standing in the pinch rather than in the open room. Then let the nearest giant rat approach; because only one neighbor of the standing tile is walkable, the pack must funnel through that single opening, so at most one rat can become adjacent at a time. Once exactly one giant rat is adjacent, attack it with the +1 long sword already in hand. After each exchange, re-check how many monsters are directly adjacent: if the intended target remains the only adjacent monster, keep attacking it. If a second or further rat is somehow adjacent despite the chosen tile (meaning the tile did not actually constrain the approach), do not stay and trade blows — relocate to a stronger chokepoint with zero or one walkable neighbor and repeat, instead of retreating in a straight line through open ground as the parent did. Continue this locate / secure-chokepoint / single-target-melee cycle until `find_all` reports no giant rats left on the level.

**N6. Description:** Add an HP-based survival rule to the melee loop: break off at low HP, get out of contact, regenerate, and only then re-engage.

**Plan:** Keep the parent's loop as the default behavior: query the map for all remaining giant rats, pick the nearest as the current target, step to its adjacent tile, and strike with the already-wielded +1 long sword, stepping back out whenever more than one rat is directly adjacent. Layer an HP gate on top of that loop. Before committing to another attack on any turn, read the agent's current Hitpoints and Max Hitpoints from the attributes; the maximum is 16 and a giant rat bite costs only a few points, so treat HP at or below about a third of maximum ($\leq$5 of 16) as the break-off point. The instant HP is at or below that threshold, stop attacking that turn — do not trade another blow. Instead withdraw: each turn step directly away from the nearest adjacent rat (the direction opposite to it), re-checking adjacency and HP after every step, and keep retreating as long as any giant rat stays adjacent. Because a pursuing rat matches the agent's speed on open floor, aim the retreat at the wall/corridor neck around (10,34) so pursuers must come through single-file and the agent can actually put a tile of separation between itself and the cluster; if rats remain adjacent turn after turn despite retreating, keep moving away rather than turning to fight. Once no giant rat is adjacent, stop approaching the cluster entirely and spend turns holding position at that distance, re-reading HP each turn, until Hitpoints recover to a safe margin well above the threshold (target near the 16 maximum, and at minimum several points clear of it). Only when HP has recovered does the agent return to the parent loop — re-locate the nearest remaining rat and re-engage single-file — and if HP again drops to the threshold, break off and recover again. Never begin or continue a melee exchange while HP is at or below the break-off point.

**N7. Description:** Once the rats are gone, apply the same one-at-a-time melee loop to the gnome still on the level, since the task requires eliminating every monster.

**Plan:** Start from the tile where the last giant rat died, at 8/16 HP. Instead of declaring the level clear, query the map for living monsters of a type other than the now-eliminated giant rat — concretely a scan for 'gnome', because the gnome corpse at (10, 48) is direct evidence that gnomes are present on this level. If the scan returns no live gnome, this stage is already done. Otherwise take the nearest live gnome as the current target and close on it; while traveling, step only onto tiles the map reports as safely walkable, deliberately skirting hazardous terrain such as bear traps, since at 8/16 HP a single trap hit can be fatal. Before trading any blows, position with a wall or a corridor tile at the agent's back so that only the target gnome is adjacent — the same single-adjacency discipline that made the rat loop work. Attack the adjacent gnome with the already-wielded +1 long sword. After each exchange, re-check which monsters are directly adjacent: if the gnome is the only adjacent monster, keep attacking it; if a second monster (gnome or otherwise) has become adjacent, retreat one step so that at most one can follow and stay adjacent, then resume attacking that single follower. Repeat this locate / travel-safely / single-adjacency / melee / disengage cycle until the map reports no live gnomes left on the level, so the non-rat monster threat is eliminated with the same loop that handled the rats.

**N8. Description:** Scout the corridor east of the room before fighting, to find other monsters and hazards such as bear traps.

**Plan:** Starting from the agent's position at (12,29), move toward the corridor heading east around (10,33), checking each step with map.get and `map.is_walkable(unsafe=False)` to avoid stepping onto any tile described as a trap (e.g., 'bear trap') or otherwise hazardous. Follow the corridor east and into any newly seen rooms or passages, systematically sweeping unexplored tiles by following walls and corridors while staying on safe, walkable floor. While moving, use `map.find_all` to detect any monsters not already visible from the starting room—note their types and approximate positions. If a monster is encountered and blocks the only safe path, engage it using the proven tactic of fighting it one-on-one (retreat to a corridor or chokepoint if other monsters could join). If at any point HP drops to or below roughly half (about 8 HP) due to combat or a trap, retreat back to the starting room or another already-cleared safe tile and wait (rest) until HP regenerates above that threshold before continuing. Stop the sweep once the immediately accessible unexplored corridor and adjacent rooms have been checked, or a monster makes further progress unsafe.

**N9. Description:** Fight the rats with a westward retreat lane toward the starting area, treating the trapped eastern dead end (where the earlier run died) as off-limits.

**Plan:** Using the completed sweep's map, hold two facts: the corridor WEST of the agent leads back through the junction (˜10,35) to the starting-room floor and stairs at (12,29), while the corridor EAST beyond ˜(10,47) dead-ends and its tile ˜(10,48) is a known bear trap (with a human corpse and a whistle on it). Treat the entire eastern dead-end, and especially the trap tile, as off-limits: never step east onto it and never select an eastward move as a retreat step. Then turn and fight the giant rats that are trailing in the corridor west of the agent: strike the single adjacent rat with the wielded +1 long sword, using positioning so at most one rat can reach the agent, and after each kill use `find_all` to confirm the monster count actually dropped, so partial progress is verified rather than assumed. Keep the westward lane open throughout — if a rat sits between the agent and the room, kill it first or step around it so the agent is never cut off from the west. Whenever HP falls to about half ($\leq 8$), stop attacking, move WEST (never east) along the corridor to the already-cleared room floor near the stairs at (12,29), and wait there until HP regenerates above that threshold before returning to the rats.

**N10. Description:** Before the rat fight, search the eastern dead end for a hidden door from adjacent floor, never stepping on the known bear trap, to make sure no monsters remain unseen.

**Plan:** Treat the known bear trap (˜(10,48)) as a permanent no-go: before entering any tile, check map.get(x,y) and `map.is_walkable(x,y,unsafe=False)`, and refuse to step onto any tile whose description contains 'trap'. From the current position, move east down the corridor toward the dead end, stopping on the last safe floor tile that is adjacent to the terminal dead-end region rather than entering the bear-trapped tile, and confirming along the way that at least one tile not visible at t=0 has been visited. Before each move while heading east, check adjacency for giant rats

with `find_all('rat')` within a small radius; if a rat is orthogonally adjacent (melee range), do not continue east — take a step back west down the corridor, and if the only westward step is itself blocked by a rat, attack that rat in melee rather than stepping onto a hazard. Once at the safe vantage next to the dead end, apply the search action several times, aimed at the terminal wall tiles, and read the message for 'You find a hidden door' or the appearance of a new passage. If a hidden door or passage is revealed, record its location and stop this segment (exploring it is a later stage); if after several search attempts nothing is found, conclude the corridor is a true dead end and stop. Throughout, if HP drops to roughly 8 or below from rat bites during the approach or the search, abandon the search, fall back west down the corridor away from the rats, and stop — without resting while any rat is adjacent.

**N11. Description:** Hold the corridor mouth, which admits one attacker at a time, and fight the whole rat battle from there, retreating east along the lane (short of the bear trap) to recover HP.

**Plan:** From the starting position (12,29), move east/northeast toward the single corridor-mouth tile in the room's east wall around (10,33), staying on tiles already known to be open room floor and never crossing the rat cluster, which sits to the north/west at roughly (8,26)-(10,29). Before each step into a tile not already verified this run, read its description and check its walkability with the non-hazard option, refusing any tile described as a trap or hazard, so the agent does not step onto a hidden bear trap; the rats are west/north of the agent while the doorway is east, so the move itself should not require entering their squares. Once at the mouth, confirm the lane east of it is one tile wide and identify the known bear trap and corpse at the far terminal end near (10,48), treating that tile as a hard boundary that is never entered and not even used as a step. Then stand at the mouth (or one tile east into the lane) and let the giant rats come: each turn, attack the single adjacent rat with the wielded +1 long sword, and because the lane is a single tile wide, only one rat can occupy a tile adjacent to the agent at a time. If a second rat reaches and occupies a tile next to the agent (or its HP drops to about half, roughly 8, from bites), step one tile east down the lane to break melee, checking that next lane tile for traps before entering and never advancing past the safe limit short of the (10,48) bear trap; the pursuing rats must now string out into the lane, restoring the one-on-one condition. Only wait to regenerate HP when no rat is adjacent; never rest while a rat is next to the agent and never back west into the open room while two or more rats are near, since that is the exact situation that produced the parent run's death.

### D.2 CORRIDOR-R3

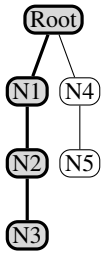



Figure 7: Scenario tree of a successful run on Corridor-R3 (task: locate and reach the down staircase). Shaded nodes with bold edges mark the path to the best-performing scenario (N3).

**N1. Description:** Leave the closed-off starting room and explore the connected corridors and rooms until the down staircase is located.

**Plan:** Start by reading the agent's own position from attributes, then inspect the four orthogonal neighbor tiles and, if none are open, walk the room's perimeter with get()/`is_walkable()` to find the doorway or gap in the wall. Move tile-by-tile toward that opening using `direction_to_string`, stepping out into the corridor beyond and confirming the agent is now on a walkable tile outside the starting room's walls. Once outside, repeatedly call `find_all()` with whole-word queries like 'staircase' over a generous radius to check whether the down staircase is already within view. While no staircase is in range, expand exploration outward: move in straight runs down corridors to their ends, and at each junction or newly entered room pick an unexplored

branch, favoring directions that lead away from corridors and dead ends already traversed, until a down-staircase tile is detected. Finish this stage by moving the agent onto a walkable tile immediately adjacent to the detected down staircase, without yet stepping onto the staircase tile itself.

**N2. Description:** Fix stair detection (stairs appear as "stairs", not "staircase") and remove the boulder blocking the west corridor, so exploration can continue to the down staircase.

**Plan:** First verify stair recognition: while near the known up stair (the tile the agent starts on, described as 'stairs up'), run `find_all` with the whole-word phrase 'stairs' over a generous radius and confirm it matches that tile, establishing that the phrase family 'stairs' (not 'staircase') is what the map uses for stair tiles. At every subsequent position re-run that 'stairs' query, and classify a match as the down staircase only when its description indicates a descent (e.g. contains 'down'), so the already-recorded up stair is never mistaken for the target. While no descending stair is in range, resume outward exploration from the current spot, moving in straight corridor runs and picking unexplored branches at junctions. The reachable area is capped at about 26 tiles because a boulder sits at the west end of the corridor west of the starting room; when exploration reaches that dead end, walk to the walkable tile adjacent to the boulder and push it by moving into it westward, clearing the corridor so travel can continue. If the boulder will not move after a couple of attempts, treat the wall tiles immediately abutting it as the one plausible hidden continuation and perform a single bounded secret-door search there, rather than a full-level sweep. Once the corridor is clear, press the outward exploration into the newly opened area, again re-querying 'stairs' after each move, until a descending-stair tile is detected. Finally, move the agent onto the walkable tile immediately adjacent to that down staircase, without stepping onto the stair tile itself. Throughout, never fall back to repeated 'open door' commands or whole-level secret-door sweeps when a move is blocked.

**N3. Description:** Instead of declaring exploration finished at the boulder, push it west to reopen the corridor, breaking the loop of secret-door searching and repeated door-opening.

**Plan:** When outward exploration reaches the west dead-end of the corridor (the walkable tile immediately east of the boulder at (7,66), where the policy currently reports 'exploration exhausted'), treat that adjacent boulder as the barrier and suppress every fallback: do not run a whole-level 34-wall secret-door sweep and do not issue any 'open door' command. Instead, confirm the agent is standing on the walkable tile directly east of the boulder so the boulder is the tile immediately to the west, then issue a single westward movement directly into the boulder tile; in this game pushing a boulder is done exactly by moving into it in a cardinal direction. Inspect the resulting message/tile change to confirm the push succeeded — the boulder should now occupy the tile one further west and the agent should be standing on the tile the boulder just vacated. Step onto each newly vacated tile and re-run the whole-word 'stairs' query to check whether a descending stair has come into range, and repeat the westward push a bounded number of times (only a few attempts), since the knowledge warns a boulder becomes immovable after enough pushes. If the boulder genuinely will not move after that small bounded number of attempts, treat only the wall tiles directly abutting the boulder as the single plausible hidden continuation and perform one short secret-door search there. In no case re-enter the exhausted scan or the repeated 'open door' loop when a move is blocked.

**N4. Description:** Find the starting room's actual exit and open it the right way, using the carried lock pick on the locked door instead of repeatedly issuing the open command.

**Plan:** Query the tiles forming the room's perimeter (the wall ring around the agent's position) with the map interface and look for the one described as a door or doorway rather than as plain wall — this is the room's opening, and scanning each perimeter tile this way is what replaces the ineffective blind wall-searching. Once the door tile is found, read its state: whether it is open, closed, or locked (an open-door description versus a closed/locked one, or a message confirming locked). If it is closed but not locked, issue a single open action on it. If it is locked or the open attempt has no effect, do not retry open — instead select the carried lock pick (inventory item 'e', the uncursed lock pick) and use it to pick the door, falling back to a single kick only if the lock pick is somehow unusable. After the door opens, step onto and through that tile to move out of the starting room into the adjacent corridor, confirming the agent is no longer inside the sealed room. If the perimeter scan turns up no door/doorway tile at all, re-examine the wall ring for a possible hidden opening rather than treating the absence as a dead end, but keep this to a targeted check rather than an exhaustive search.

**N5. Description:** Once out of the room, explore unvisited parts of the level without looping until the down staircase is located.

**Plan:** Begin from the corridor tile just outside the starting room's door — the tile the parent scenario ended on — and treat that tile plus everything already seen as visited. Each step, look at the walkable tiles adjacent to the current position and move into one that has not been visited yet (an untried corridor branch or a newly-adjacent tile), and never step straight back into the tile just left. Keep the set of visited tiles growing: if the agent's recent moves have revealed no new tile, stop bouncing between the same few corridor tiles and instead backtrack along the corridor to the nearest junction that still has an unvisited branch, then take that branch; do not fall back on exhaustive wall/secret-door searching, which only helps when there is a high-probability hidden exit. While moving, periodically query the map interface for tiles whose description matches a staircase (whole words such as 'staircase' or 'down'), so that as soon as the down staircase enters view its coordinates are recorded and it becomes the exploration target. When a corridor is blocked by a closed or locked door, read the door's state and open it with the method already established for the starting room — a single open on a closed unlocked door, the carried lock pick on a locked one — entering through the doorway with a cardinal move rather than a diagonal; when a boulder blocks a corridor, push it rather than treating the passage as a dead end. Continue this until the down staircase has been located and its position recorded.

### D.3 QUEST-EASY

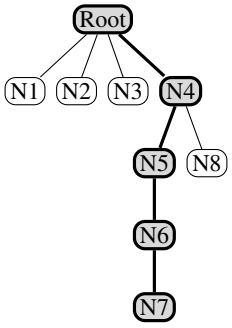



Figure 8: Scenario tree of a successful run on Quest-Easy (task: cross the lava barrier and descend the staircase). Shaded nodes with bold edges mark the path to the best-performing scenario (N7).

**N1. Description:** Walk to the visible down staircase to the east, detouring north or south around the lava band that blocks the direct route.

**Plan:** Locate the down-stairs tile with `find_all` by searching for 'stairs'/'staircase', confirming it is the eastern staircase about 24 tiles east and 1 tile south of the agent's current position. Then identify the hazard: query tiles along the eastward line (or use `find_all` for 'lava') to note the lava row spanning roughly x=8..12 at y=30 that blocks a straight eastward walk from the agent's row. Choose a detour: before reaching that lava band, shift north (toward x¡8) or south (toward x¿12) to whichever side offers a walkable crossing, verify each candidate tile with `is_walkable`, cross the lava's latitude at that offset, then resume heading east toward the stairs' y-coordinate. Re-check walkability on every step and never step onto a lava tile. Treat the newt at (9,29) as a minor obstacle only: if it stands on the chosen path, step around it when a walkable tile is available, and only if it directly blocks a one-tile corridor, kill it with the wielded +1 long sword rather than chasing it. Keep re-locating the down-stairs each turn and pressing eastward (adjusting north/south as needed) until the agent is standing on the down-stairs tile.

**N2. Description:** Move sideways along the short lava wall until its end is rounded and a lava-free path east opens, stopping at that crossing point.

**Plan:** Read the agent's own position and confirm the down-stairs are to the east (larger east-west coordinate), with lava occupying the tiles directly east along the agent's current north-south column. Recognize this lava is a finite segment — a handful of tiles wide along one north-south span at a

single east-west line — so it cannot be crossed and must be flanked, and waiting in place is never an option. Inspect the tiles one step north and one step south of the agent to see which lateral direction stays walkable and leads toward the nearest end of the lava segment, and choose that direction. Step one tile at a time in the chosen lateral direction, staying on walkable tiles; if the next tile in the chosen direction is not walkable, immediately try the opposite lateral direction instead, and keep moving — never remain on the same tile while blocked. After each lateral step, check whether the agent's current north-south position is now outside the lava segment's span and whether the tile immediately east is walkable, i.e. whether a lava-free eastward route now exists at this row. Stop as soon as the agent stands on such a tile: it has rounded the wall and opened the eastward path toward the stairs.

**N3. Description:** The lava fills the width of the room, so search its other walls for a non-lava exit and step out through it.

**Plan:** Treat the lava at (8,30)-(12,30) as an impassable wall and stop trying to walk east through it. From the current position (9,26), survey the boundary of the starting room tile by tile, checking the north, south, and west walls as well as the east edge that flanks the lava, and identify the first boundary neighbor that is walkable and is not lava. Move to that opening and step through it into the adjacent corridor, leaving the starting room's band, taking care never to set foot on a lava tile. Stop as soon as the agent is standing in a walkable tile outside the starting room, confirming that a non-lava way out exists.

**N4. Description:** Since walking cannot get past the lava wall, zap the carried unidentified wand into the lava and observe its effect (e.g., freezing, digging, or teleportation).

**Plan:** The agent is at (9,26), the lava wall spans rows 8-12 at column 30, and the down-stairs lies beyond at (10,50). First make the lava's western edge reachable: the newt sitting at (9,29) is directly west of a lava tile on the direct line, so engage and kill it with the already-wielded +1 long sword (a level-1 newt is trivial), or, if that is slower, step to a different walkable tile that sits immediately west of any lava tile in the wall — e.g. (8,29), (10,29), (11,29), or (12,29). Once the agent is standing on a walkable tile directly adjacent (west) to a lava tile, aim the steel wand eastward into the lava and zap it. Then read the resulting message and inspect the targeted lava tile to determine what the wand actually did: whether it froze/turned the lava solid, dug or otherwise altered terrain, teleported or displaced the agent, or produced 'nothing happens'. The stage ends the moment the wand has been fired at the lava and its effect is observable — no crossing or descent is attempted yet.

**N5. Description:** The wand turned out to be a wand of cold that solidified a lava tile; kill the rat sitting on the new crossing and step through to the east side of the wall.

**Plan:** The agent is standing at (9,29), directly west of the now-solidified gap at (9,30); beyond the wall the route continues east toward the down-stairs at (10,50). First deal with the sewer rat occupying (9,30): engage it with the already-wielded +1 long sword and attack until it is dead (a single 1st-level rat is trivial for the agent's 16 HP and AC 3). If the rat steps off the breach tile on its own, no attack is needed. Once (9,30) is clear, move east onto the solidified tile (9,30) and then take one more east step to a walkable tile on the east side (e.g. (9,31)) so the agent is standing past the wall rather than perched on the single breach tile. Do not yet try to reach the down-stairs or fight the kobold zombie further east — this stage ends the moment the agent has crossed to the east side of the lava wall.

**N6. Description:** From the east side of the lava wall, move along the corridor to the down staircase, killing the jackals that block the way.

**Plan:** From (9,31), first locate the down-stairs: query the map for the stairs-down tile (it is not visible in the current view) to confirm its position, so the agent heads toward the real target instead of a guessed coordinate. Then move east along the walkable corridor (roughly row 9-10) one tile at a time. The jackals at (9,38) and (10,39) occupy the corridor and block a straight eastward path; engage the nearest blocking jackal with the already-wielded +1 long sword and attack until it is dead, then continue east. If a jackal steps off the corridor tile on its own or an adjacent walkable tile is open around it, simply step around it instead of waiting or fighting. Keep advancing east toward the located down-stairs tile until the agent is standing on it or on a walkable tile directly adjacent to it. Do not step onto the stairs and descend yet — that is the next stage.

**N7. Description:** Standing next to the down staircase, step onto it and descend to the next dungeon level.

**Plan:** From (10,49), step east onto the down-stairs tile at (10,50). Before that step, confirm the down-stairs tile is free of any monster: the grid bug last seen near (10,48) (and any other entity) must not be occupying (10,50), since a monster on the stairs would block the descent — if one is sitting on the stairs tile, engage it with the already-wielded +1 long sword and attack until it is dead or steps off, rather than waiting. Once the agent is standing on the down-stairs tile (10,50), issue the descend command (the '>'-style descend action) and read the resulting message to confirm the descent occurred. This stage ends the moment the agent has descended — no further movement, combat, or exploration on the new level is attempted yet.

**N8. Description:** With the lava crossed, advance along the east corridor to the down staircase, killing the kobold zombie that blocks it.

**Plan:** Starting from the east side of the solidified breach (around (9,30)-(9,31)) on dungeon level 1, move east along the walkable corridor toward the down-stairs at (10,50), recomputing the walkable path each turn and staying off any remaining lava (the surviving lava tiles (8,30), (10,30), (11,30), (12,30) are all west of the breach and are not on the eastward route). If the kobold zombie ahead in the corridor (last seen around (9,35)-(10,36)) blocks the way, engage it with the already-wielded +1 long sword and keep attacking until it is dead, then continue east. Keep advancing east/northeast through the corridor until the agent is standing on or immediately adjacent to the down-stairs at (10,50). The stage ends the moment the agent reaches the stairs; no descent is attempted yet.

### D.4 WoD

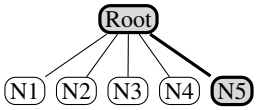



Figure 9: Scenario tree of a successful run on WoD (task: kill the troll). Shaded nodes with bold edges mark the path to the best-performing scenario (N5).

**N1. Description:** Attack the troll from range with the +2 darts from lava-free ground, since the fragile level-1 Tourist would likely die in melee.

**Plan:** Query `map_interface` for the troll's current tile to confirm its position (initially (12, 33)) and re-check that position every turn, since the troll can move toward the agent. Keep the agent on walkable non-lava tiles at all times: the troll sits in a narrow band bounded by lava at (12, 30) and the (9, 30-46)/(13, 30-46) columns, so never step onto a lava tile while repositioning or retreating. From a clear position along the open corridor between the agent (11, 43) and the troll, throw a +2 dart in the direction of the troll's tile, one throw per turn, re-querying the troll's tile before each throw and re-aiming at its new tile if it has moved. Continue throwing at the troll until the game message confirms it is dead, or until the troll reaches a tile adjacent to the agent, in which case the agent first steps to a walkable, non-lava tile that increases its distance from the troll and then resumes throwing, rather than staying in melee.

**N2. Description:** Pick up the unidentified jeweled wand next to the agent and zap it in a safe direction to learn what it does, staying out of the troll's reach.

**Plan:** From the current position at (11,43), step onto the jeweled wand tile at (11,42) (one step north, adjacent) and pick the wand up so it is added to inventory. With the jeweled wand in inventory, determine what it is: since it is unidentified, probe it by zapping it in a safe direction that does not point toward the troll at (12,33), and read the game message that reports the effect, so its type and behavior are known. Keep the troll at a distance throughout this segment — do not advance toward (12,33) or let the troll close while the wand is being retrieved and tested, so that the agent learns what the wand does without being exposed to a melee hit or a death-ray zap.

**N3. Description:** Blind the troll with the camera flash (or the cream pie as a backup) from a non-adjacent tile, so it cannot aim a wand of death ray and fights less accurately.

**Plan:** From (11,43) with the troll at (12,33) roughly ten tiles west and one south, move west along the lava-free floor between the x=9 and x=13 lava columns, closing only until the troll is within the expensive camera's flash range. Stop at least two tiles short of the troll's tile so it cannot melee the agent. Then apply the expensive camera and aim it directly at the troll's tile so its flash blinds the troll. Read the resulting game message to confirm the troll is blinded — a blinded troll cannot aim a death-ray wand and swings much less accurately. If the camera flash does not blind the troll, throw the uncursed cream pie at it from the same non-adjacent range as a backup blinding attack. If neither blinding attempt works, withdraw west down the corridor rather than remain in the troll's line of fire. Never step onto a lava tile or into a tile adjacent to the troll.

**N4. Description:** Wear the troll down with darts while kiting, keeping out of melee and off its firing line, and drink extra-healing potions at low HP.

**Plan:** From the current position (11,43), first confirm the agent is standing on solid, non-lava ground and that the troll at (12,33) is at least about five tiles away; if not, spend turns stepping along the solid walkway away from the troll, never entering a lava tile, until that separation is restored. With the +2 darts already at the ready, throw one dart at the troll on any turn in which the troll is not on a tile orthogonally or diagonally adjacent to the agent. Immediately after each throw, move one or more steps away from the troll along solid non-lava ground to reopen the gap to roughly five or more tiles; this both keeps the troll out of melee range and pulls the agent off the straight line the troll would need to aim a wand, so a death ray cannot connect. If the troll ever closes to an adjacent tile before the agent's turn, do not attack from that adjacency (a troll in melee will kill a 10-HP character) — spend that turn disengaging to solid, non-adjacent ground instead, and only resume throwing once the standoff is re-established. Track Hitpoints: if a hit lands and HP falls to 4 or lower, quaff one of the two uncursed potions of extra healing before continuing the throw-and-retreat loop. Keep repeating the throw, reposition, re-check-distance, and re-check-HP cycle until the message reports the troll is dead, confirming lava tiles were never stepped on at any point.

**N5. Description:** Pick up the pine wand one step away (possibly a wand of death) and zap it at the troll from a safe, non-adjacent tile, learning what the wand does at the same time.

**Plan:** Move onto the tile one step north at (11,42) and pick up the pine wand so it is in the agent's inventory and can be zapped. Then move to a firing position that keeps the fragile Tourist safe: at least two tiles (not adjacent) from the troll at (12,33) so a blinded-or-not troll cannot melee, and on a clear straight line to the troll so the wand's beam is not blocked by intervening lava or wall tiles, staying off the lava tiles that border the corridor. From that position, apply the pine wand aiming the zap along the straight direction toward the troll (roughly west from the agent's start area), so the beam is directed at the monster. Immediately read the resulting game message to learn what the wand actually does — whether it kills the troll outright (e.g. a death ray), merely damages it, teleports it away, or has no effect on it — recording that result as the factual basis for whatever damage step comes next.

### D.5 KEYROOM

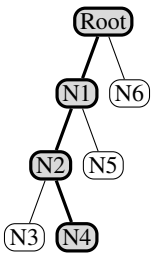



Figure 10: Scenario tree of a successful run on KeyRoom (task: find the key, get past the locked door, and descend the staircase). Shaded nodes with bold edges mark the path to the best-performing scenario (N4).

**N1. Description:** Walk down the south corridor and pick up the key visible on its floor, which also reveals more of the map.

**Plan:** From the starting position (6, 36), walk south step by step along the open corridor (increasing x while keeping y = 36) rather than wandering into unexplored side tiles, so the agent stays on the known-walkable path. Before each step, confirm the target tile is walkable; if the corridor is blocked or the tile is not passable, stop and treat the corridor as a dead end rather than forcing through. Continue south until reaching the tile the map legend marks as 'key' at (15, 36). When the agent is standing on that tile and the game presents the item (via the message text or a pickup prompt/choices), take the key. This single southward sweep is the stage: it collects the key and extends explored map southward.

**N2. Description:** With the key in hand and the starting room explored, open the room's only remaining closed door to reach the unexplored area beyond.

**Plan:** From wherever the agent currently stands inside the fully-mapped starting room, walk tile-by-tile to the closed vertical door at (9,39), staying on tiles confirmed walkable. Critically, avoid the two-tile oscillation that dominated the parent run: never step back onto the tile the agent just came from in the immediately preceding move, and prefer moving toward (9,39) over any candidate that would revisit recent tiles. Because the room interior is open floor, this is a straightforward approach; if a monster or item blocks the path, route around it rather than stopping. On arriving at the door tile, open the closed door (walking into it is the standard way to open a closed door), which reveals the tile(s) beyond and reports the result in the message. If the door merely opens, that is the success of this stage — the newly revealed area becomes the target for the next stage. If the door is locked and does not open, note that as the outcome to build on.

**N3. Description:** Unlock the locked door by applying the lock pick and actually selecting it at the item prompt, rather than cancelling the prompt and stalling.

**Plan:** From wherever the agent stands, make sure it is on a tile orthogonally adjacent (not diagonal) to the locked door at (7,39) — move orthogonally toward it if needed, never diagonally into the doorway. Issue the apply / use-tool action targeting the lock pick rather than a bare walk into the door, since walking into a locked door can never open it. When the game presents the inventory-selection prompt asking which item to use, select the uncursed lock pick the agent is carrying (item 'e'); if the agent happens to be carrying a key such as The Master Key of Thievery, applying that to the door is the equivalent alternative. Read the message the game returns after the selection: a lock-picking success is a message indicating the lock turns or the door unlatches/opens, and a failure is a message like the pick slipping — either way the outcome must be taken from that message, not assumed. Throughout, guard against the parent's stall: if an action produces no state change (the same message and the same position as the immediately preceding step, e.g. 'Never mind.' or a repeated 'This door is locked.'), stop repeating it and switch to the lock-pick selection path instead of re-issuing the same bare action indefinitely. The stage succeeds once the door is unlocked/opened and the tiles beyond it are revealed.

**N4. Description:** Try the other unlocking tool, the Master Key of Thievery, on the locked door, recording the result without stalling.

**Plan:** The parent stage got the agent up against the locked door at (7,39) but could not get through, and the lock-pick approach is already covered by a sibling branch. This stage instead tries the agent's other unlocking tool — The Master Key of Thievery — on that same door. First, from wherever the agent stands, walk tile-by-tile while staying only on tiles confirmed walkable until it occupies a tile orthogonally adjacent to the door at (7,39); never attempt to enter a doorway diagonally. Confirm the tile is the locked door by the message 'This door is locked.'. Then consult the inventory list to locate The Master Key of Thievery and note the letter it occupies; if the key is not actually carried, record that it must be retrieved before this attempt and do not issue an apply. With the key confirmed in hand, trigger the apply/use-tool action and, when the game presents the item-selection prompt, answer it with the key's inventory letter so the attempt actually executes instead of cancelling with 'Never mind.'. Read the resulting message: if the door unlocks or swings open, the area beyond is revealed and that is this stage's success; if the door remains locked or the action again cancels, record that outcome and stop rather than repeating the same no-progress action.

**N5. Description:** Explore the whole starting room without backtracking, fixing the loop that trapped the agent after the pickup and confirming the room's walls, up staircase, and single door.

**Plan:** Starting from the open floor tile the agent occupies after collecting the key inside the starting room, switch from steering toward a fixed compass target to a single consistent sweep rule: at each step, prefer an adjacent walkable floor tile that has not been stood on yet, choosing the continuation that keeps the agent moving along the wall or in the same general heading. Never step back onto the tile the agent just came from while at least one unvisited walkable neighbor still exists. Hug the room's perimeter where possible, since doors and staircases sit on the walls, then fill in the interior tiles. Continue the pass until every walkable floor tile reachable from the start inside the room has been visited at least once, or until a new feature (a down staircase, another door, an item, or a monster) enters view; when that happens, stop the sweep and record that feature's location as the next lead. Do not open the door, descend, or fight anything during this stage — the sole goal is dependable coverage.

**N6. Description:** Open the locked door near the start by applying the lock pick and selecting it when the game asks for an item.

**Plan:** Move from the current position to a tile orthogonally adjacent to the closed/locked door at (9,39), using only north/south/east/west steps because diagonal movement into a doorway is not possible. Then apply the lock pick: trigger the apply action and, when the inventory selection prompt appears asking which item to apply, select the uncursed lock pick (the 'e' entry) rather than confirming a default, since the bare apply sequence does not handle this prompt. When the game then asks for a direction, aim the lock pick at the door and attempt to pick its lock. If the door still reports locked, repeat the apply-and-select sequence and target the door again. Once the door opens, treat the doorway as now passable for future movement.