UP AND DOWN THE ABSTRACTION LADDER: CODE-BASED SKILLS FOR LANGUAGE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Language agents struggle to act and learn in environments that require long sequences of low-level actions. Code-based abstractions can make these agents more productive by letting them invoke reusable skills instead of repeatedly selecting individual actions. The code handles recurring local decisions, while the language model decides which skills to use and how to combine them. Yet abstractions are leaky, and situations beyond a skill's capabilities may require a return to primitive actions. Motivated by this tradeoff between productivity and flexibility, we systematically study how code-based action abstraction affects the performance, inference cost, and learning of language agents. We study this in NetHack, a challenging, long-horizon game environment, using `CodeHack`, our library of code-based skills with natural-language descriptions. We use this library to compare agents restricted to primitives with those using semantic skills alone or in combination with primitives. We evaluate these agents in three settings: zero-shot prompting, supervised fine-tuning, and reinforcement learning. Across a broad zero-shot evaluation on NetHack, we find that compared with primitives, skills nearly triple game progression, while reducing inference cost per episode by 86%. Combining skills with primitives retains much of this benefit while preserving a path back down to low-level actions. Finally, in RL, we find that skill-based agents learn significantly faster than agents acting on primitives, achieving a 7.2x larger average gain in dungeon level over the same training budget. These results show that a supplied skill library can improve performance, efficiency, and learning, while retaining primitives provides flexibility when the library is insufficient. We release `CodeHack` together with training and evaluation code. †

1 INTRODUCTION

Language agents are increasingly tasked with complex, long-horizon problems, ranging from software engineering (Yang et al., 2024) and computer use (Xie et al., 2024; Yuan et al., 2026) to embodied control (Liang et al., 2023). Many of these environments require chaining together long sequences of *low-level*, primitive actions, such as precise mouse clicks and keystrokes for computer use agents, or chains of `cd` and `ls` commands for software engineering agents. While operating on this primitive action space allows for the finest level of control, it can also be costly when each small step requires a separate model call. Elevating the action interface to a higher level of temporal abstraction can help facilitate learning and exploration (Sutton et al., 1999; Nachum et al., 2019).

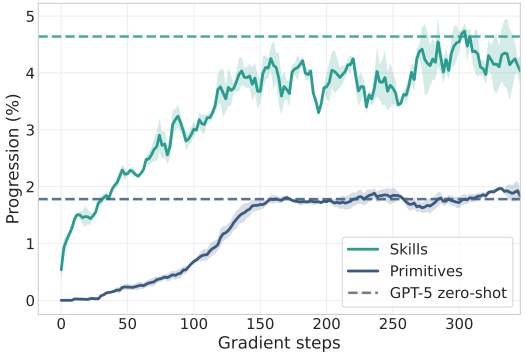

Figure 1: **The value of skills vs. primitives for language agents in NetHack.** Higher-level, semantically meaningful skills provided by `CodeHack` improve performance of `Qwen-3.5-4B` both in zero-shot use and during RL.

†Our title is inspired by Bret Victor's essay "Up and Down the Ladder of Abstraction."

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.

Past approaches have addressed these challenges through agent-computer interfaces (Yang et al., 2024; Wang et al., 2024) and temporal abstractions (Sutton et al., 1999). However, to the best of our knowledge, no study has examined the empirical benefits and tradeoffs of using primitive actions, higher-level skills, or a mixture of both for language agents in long-horizon settings. This paper aims to fill that gap by studying the use of temporally extended code-based actions, or "skills." Skills, closely related to options in hierarchical reinforcement learning (Sutton et al., 1999; Bacon et al., 2017; Klissarov et al., 2025a), execute sequences of primitive actions behind a semantically meaningful interface. Rather than learning the skills themselves, we ask how useful a fixed library of such skills is for language-model agents and learned controllers. To make this comparison possible, we build `CodeHack`, a code-based skill library for NetHack and MiniHack that exposes the same environment through primitive-only, skill-only, and mixed action interfaces. Code-based skills are natural for language agents because they can be described in language and inspected as source code. They also open a path toward agents that construct and refine these skills themselves (Liang et al., 2023; Wang et al., 2024).

We first compare skills with primitives in MiniHack, where controlled tasks let us isolate navigation, exploration, combat, and item-use. We then use NetHack as our main domain because completing it remains a longstanding challenge, despite extensive work on RL, imitation learning, and language-based control (Piterbarg et al., 2023; Tuyls et al., 2024; Wolczyk et al., 2024; Paglieri et al., 2025). Successful games span tens to hundreds of thousands of turns, while a single mistake can end a run (Kuttler et al., 2020). The variety of situations agents can encounter in NetHack also makes pure skill abstraction incomplete in practice as no practical fixed library can anticipate every contingency, so agents may still need to move back down to primitive control when abstractions are poorly matched to the current state. We therefore also evaluate mixed control on NetHack, which allows the agent to move up and down the abstraction ladder.

Contributions. In this paper, we provide a systematic study of action abstraction for long-horizon language agents. Specifically, our work makes the following contributions:

1. **`CodeHack`.** We develop and release `CodeHack`, an open-source library of reusable Python skills for NetHack and MiniHack, together with training and evaluation code for studying primitive-only, skill-only, and mixed control.
2. **A systematic study of action abstraction.** We compare agents that use primitives, skills, or both, in zero-shot, SFT, and RL settings, and present careful ablations relevant to skill coverage.
   - **Zero-shot control.** Skills improve success rates on every evaluated MiniHack task, with an average gain of 55 percentage points over primitive control. These benefits extend to NetHack, where skills nearly triple average progression across 14 models while reducing inference cost per episode by 86% relative to primitives. Adding primitives alongside skills preserves most of these performance benefits, although at a higher inference cost than skill-only control.
   - **Learning with RL.** We study how the action interface affects learning under a fixed RL budget in NetHack. Averaged across two models, skill-only and mixed controllers achieve gains in dungeon depth 7.2 and 8.6 times those of primitive-only controllers, respectively. These larger gains widen the absolute performance gap already present in zero-shot evaluation, showing that the benefits of skills extend to learning.
   - **Incomplete skill coverage.** We investigate whether retaining access to primitive actions alongside skills helps compensate for incomplete skill coverage, and find that mixed control is generally less affected by missing skills than skill-only control. Selected trajectories illustrate how primitive fallback works in practice, with agents using primitives to resolve situations that the available skills cannot handle before returning to skills.

2 PRELIMINARIES

While skills and options have a rich history in the classic RL literature (see Section 6), they are not as commonly used for language agents. As such, we here give a brief overview of options as the formal framework that underlies our work.

In a fully observable Markov decision process (MDP), an option ω is usually defined by an initiation set $\mathcal{I}_\omega \subseteq \mathcal{S}$, an intra-option policy $\pi_\omega(a \mid s)$, and a termination function $\beta_\omega(s)$. However, in a

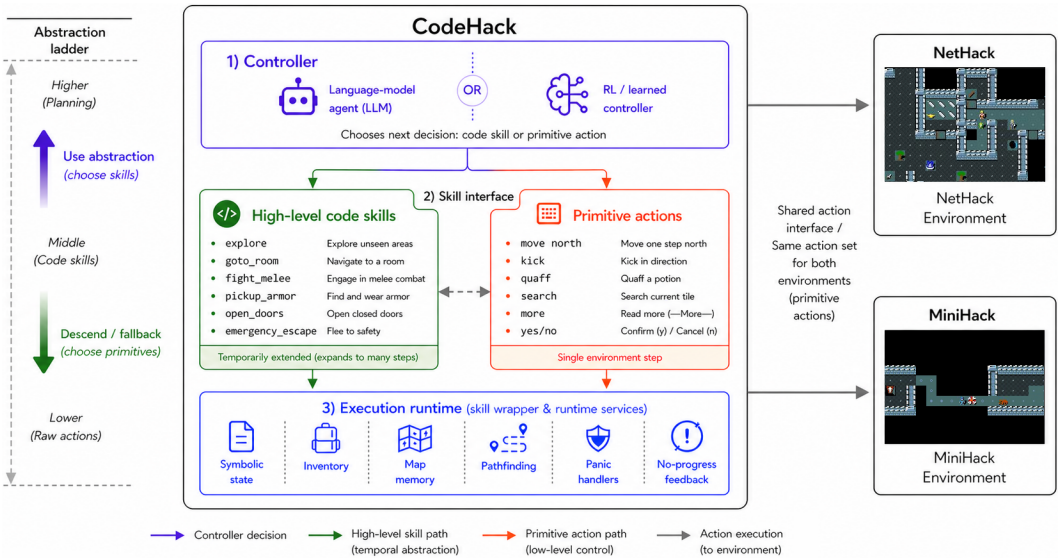

Figure 2: `CodeHack` **as a shared control layer for primitive-only, skill-only, and mixed agents. 1.** The controller can be either a language-model agent or a learned RL policy, since `CodeHack` supports both natural-language control and a discrete action-space interface. **2.** Depending on the configured action interface, the controller selects code skills such as `explore` or `fight_melee`, primitive commands such as movement, `kick`, or `search`, or both. **3.** All three interfaces use the same runtime for symbolic state, inventory tracking, map memory, panic handlers, and no-progress feedback. Primitive commands are registered as one-step strategies. Code skills can expand into multiple primitives. **4.** This runtime supports both NetHack and MiniHack, allowing the same skill library to transfer across environments while preserving a path back to low-level control when abstractions are insufficient.

partially observable MDP (POMDP) like NetHack, the agent does not observe the latent state s_t, so we define these components over the interaction history h_t. Thus, each skill ω has an initiation set $\mathcal{I}_\omega \subseteq \mathcal{H}$, an intra-skill policy $\pi_\omega(a_t \mid h_t)$, and a termination function $\beta_\omega(h_t)$. The initiation set $\mathcal{I}_\omega$ contains all histories that an option ω can execute from. In our work, $\forall \omega \in \Omega : \mathcal{I}_\omega = \mathcal{H}$ where $\mathcal{H}$ is the set of all histories. The intra-option policy $\pi_\omega(a \mid h)$ is a mapping from a history to an action to take at that history, i.e. $\pi_\omega : \mathcal{H} \to \Delta(\mathcal{A})$. Finally, the termination function $\beta_\omega(h)$ gives the probability with which option ω stops execution when at history h. In our work $\beta_\omega(h)$ will be 1 whenever the option ω has finished executing its task (e.g. killed a monster), or when the skill policy runs into an unexpected situation such as when the agent loses health points during execution of the skill. Finally, there is the high-level option policy $\mu : \mathcal{H} \to \Delta(\Omega \cup \mathcal{A})$, which outputs a distribution over all options Ω, all low-level actions $\mathcal{A}$, or both combined. In our work, the high-level option policy μ will be a language model, which will follow the call-and-return model (Klissarov et al., 2025a) to switch between high-level decision and low-level option policy control.

Remark 2.1. *Note that our notion of "skills" in this work is related to, but different from other popular LLM-centric notions of skills such as agentic markdown files (Han et al., 2026), metacognitive descriptions of reasoning procedures (Didolkar et al., 2024), or tools (Schick et al., 2023). Instead, skills are policies that operate in the same environment as the controller itself.*

3 CodeHack: A Library of Code Skills for NetHack

We build `CodeHack`, a library of code-based skills for NetHack and MiniHack together with a runtime shared by primitive-only, skill-only, and mixed agents. In all three settings, the runtime maintains symbolic state, inventory tracking, and map memory. Primitive commands are registered as one-step strategies and pass through the same execution machinery as code skills. Each code skill is a Python procedure that accesses this structured state to issue primitive NetHack actions, such as movement commands, inventory interactions, or prompt responses, and can adapt to new observations during execution. The language controller receives textual observations, including

inventory and summaries of the remembered map, with bounded interaction history, and selects the next action when control returns. `CodeHack` is not intended to solve NetHack end-to-end through a fixed hand-written policy. Instead, it provides actions that a language model or learned controller can invoke during sequential decision making. Fig. 2 summarizes this division: the controller selects an action from the configured repertoire, and the runtime executes it. The same runtime is reused across NetHack and MiniHack, enabling direct transfer of the skill library and mixed control over multiple abstraction levels. Additional technical details of the implementation are provided in Appendix A.

We focus on code-based skills because they provide a concrete substrate for studying action abstraction. Each skill has an explicit name, a readable implementation, and a behavioral specification that can be tested directly in NetHack or MiniHack. Compared with latent skills, this makes the action interface easier to inspect, modify, and extend, while giving language models access to semantically grounded actions such as exploration, combat, or item management.

The library implements a broad repertoire of higher-level skills for recurrent NetHack behaviors. These include exploration and navigation, combat and tactical control, inventory management, and recovery handling. Representative examples include `explore`, `goto_room`, `fight_melee`, `pickup_armor`, `open_doors`, `leave_shop`, and `emergency_escape`. Our NetHack experiments expose a curated repertoire of 78 skills. MiniHack configurations use subsets of this repertoire, augmented with 7 additional skills for lava crossing and levitation. The skill lists and configurations used in the primary zero-shot evaluation are provided in Appendix A.5.

This skill action space is strong but incomplete. Rather than attempting to cover all of NetHack with a fully engineered skill library, we prioritize frequent and reusable behaviors that capture a substantial portion of practical play. This makes the setting a more realistic test of skill-based control: agents benefit from abstraction, but they must still cope with missing skills, edge cases, and local failures.

4 Experimental setup

Environments We use MiniHack (Samvelyan et al., 2021) for controlled tests of specific behaviors and the NetHack Learning Environment (NLE) (Kuttler et al., 2020) for long-horizon evaluation and learning. Our MiniHack suite comprises seven tasks: `Corridor-{R3,R5,R10}` test navigation and exploration, `WoD-Hard-Full` tests acquiring and using a wand in combat, and `Quest-{Easy,Medium,Hard}` combine navigation, exploration, item use, and combat in increasingly difficult settings. NetHack requires coordinating these behaviors over thousands of steps while exploring more than 50 levels of a procedurally generated dungeon to retrieve the Amulet of Yendor and return to the surface. All NetHack experiments use `NetHackScore-v0`. Appendices B.1 and B.2 describe the environments and task configurations.

Environment setup Both domains are accessed through the `CodeHack` environment runtime, which wraps the underlying NLE or MiniHack simulator. For each task, we keep the underlying game parameters, observation format, and prompt template fixed across action interfaces, changing the available-action lists and descriptions to match the selected interface. We use the NLE language wrapper (Goodger et al., 2023) to convert observations into natural language and an ASCII map. Following the BALROG setup (Paglieri et al., 2025), we then present this information to the LLM through a chat template. At each timestep t, the agent receives a bounded interaction history together with the current observation o_t inside this chat template, guided by a system prompt that describes the task and the available actions. Appendix B.3 details the prompt structure, history settings, and handling of malformed outputs. Appendix B.1 gives the NetHack episode limits and termination settings.

Action interfaces We compare three action interfaces through the `CodeHack` runtime. Primitive-only agents select low-level game commands, skill-only agents select code-based skills, and mixed agents can select either. NetHack uses a repertoire of 78 skills, while MiniHack uses task-specific subsets of this repertoire, augmented with additional skills described in Appendix A.5. The skill implementations remain fixed throughout the experiments.

Evaluation metrics To evaluate game performance, we use task success rate in MiniHack and score, progression, and dungeon level in NetHack. The NetHack score is built into the game and takes into account a variety of factors such as dungeon depth, enemy kills, and collected gold. Since

game score is not always aligned with winning, we also report the progression metric introduced by Paglieri et al. (2025), which estimates progress from the highest dungeon and experience levels reached using a mapping derived from human trajectories (Appendix B.1). Finally, dungeon level tracks the maximum dungeon level the agent reaches anywhere in the episode. We also report token usage and estimated inference cost per episode, using OpenRouter input and output token prices.

Zero-shot evaluation We compare all three action interfaces on NetHack across 14 models from the GPT (Singh et al., 2025; Agarwal et al., 2025), Llama (Grattafiori et al., 2024), Gemma3 (Gemma, 2025), Gemma-4, and Qwen-3.5 (Qwen Team, 2026) families. GPT-5, GPT-OSS-120B, Gemma-4, and Qwen-3.5 use reasoning mode, while Llama and Gemma3 use act-only control. On MiniHack, we compare primitive-only and skill-only control using GPT-5.

To test whether primitive actions help compensate for missing skills, we remove one skill family at a time and evaluate Gemma-4-31B on NetHack. We compare skill-only and mixed control using the same reduced library, with primitive actions available under mixed control (Appendix C.1).

Learning experiments We train `Llama-3.1-8B-Instruct` and `Qwen-3.5-4B` on NetHack with PPO (Schulman et al., 2017) under all three action interfaces. Both models use act-only control in this comparison, including their zero-shot baselines. The reward is the change in game score, aggregated over the primitive transitions executed by each skill. PPO uses 32 rollouts per trainer GPU, each containing 16 controller decisions. We compare policies at a common checkpoint after 264 gradient updates and report learning curves against gradient updates and wall-clock time (Appendix D). Skills can execute multiple primitive actions per decision, so equal rollout lengths can involve different numbers of primitive environment transitions.

We also test whether supervised fine-tuning (SFT) improves initialization for skill-only RL. We collect 1,024 zero-shot teacher trajectories with `Gemma-4-31B` and fine-tune both models for one epoch under skill-only control. We evaluate the SFT policies directly and use them to initialize additional PPO runs. We compare PPO with and without SFT initialization after the same number of gradient updates. SFT adds dataset collection and supervised training to the total training cost. Appendix B.4 gives the training hyperparameters.

5 Language agents with skills: Experimental results

We organize the results around five empirical questions, which we address through six research findings (RF).

1. How do skills affect zero-shot performance across tasks and models? *(RF1, RF2)*
2. How do skills affect inference cost? *(RF3)*
3. What are the benefits and tradeoffs of allowing primitive actions alongside skills? *(RF4)*
4. How do skills affect gains over a fixed RL budget? *(RF5)*
5. Does SFT initialization improve the final RL policy? *(RF6)*

Table 1: **Summary of the 14-model zero-shot NetHack sweep.** Values are unweighted averages across models with gray standard errors across model-level means. Progression is in percent; lower is better for cost and token usage. Appendix C gives the full per-model breakdown.

Metric	Primitives	Mixed	Skills
Progression	0.69 ± 0.16	1.88 ± 0.33	1.98 ± 0.33
Score	83.9 ± 20.2	291.0 ± 71.0	318.3 ± 89.2
Dlvl	1.19 ± 0.05	2.79 ± 0.35	2.88 ± 0.34
Cost/ep. ($)	4.35 ± 3.56	1.36 ± 1.04	0.59 ± 0.45
Tok./ep. (M)	6.72 ± 1.13	3.54 ± 0.53	1.74 ± 0.31

RF1: Skills improve success rate across all evaluated MiniHack tasks. In Fig. 3 (left), we compare `GPT-5`'s zero-shot success rates using `CodeHack` skills versus primitive-only control across seven MiniHack tasks. Skills yield large improvements across all evaluated tasks, averaging 55 percentage points over primitive control. Skills achieve near-perfect success on all three `Corridor` tasks, which isolate navigation and exploration. On `WoD-Hard-Full`, success rises from 28% to 97%, extending the gains to acquiring and using a wand in combat. Skills also improve `Quest-Easy` and `Quest-Medium`, which require combining navigation, item use, and combat, although success remains below that on the `Corridor` and `WoD` tasks. Skills enable 25% success even on `Quest-Hard`, the hardest task in our suite, where primitive control solves no episodes. Together, these results

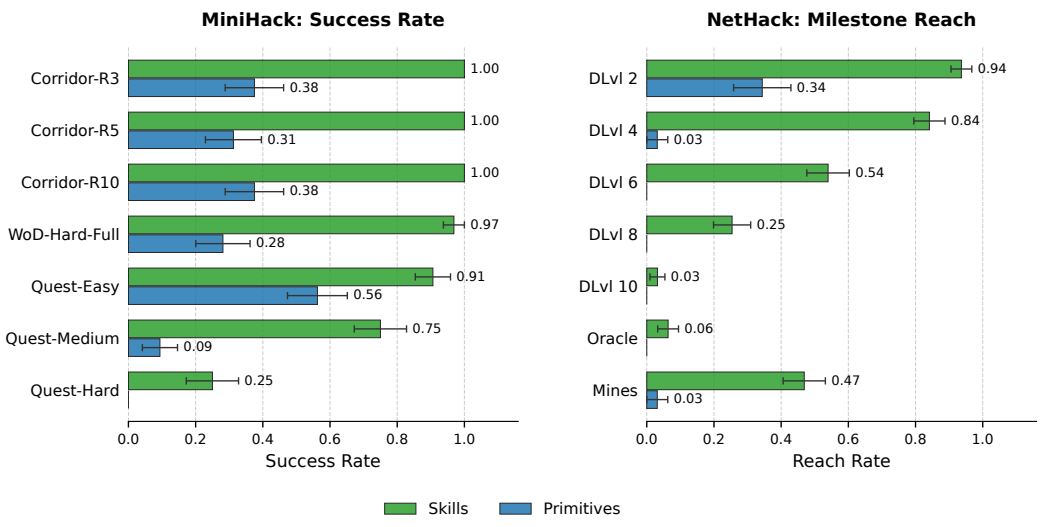

Figure 3: **Zero-shot results across the two evaluation domains.** ***(Left)*** MiniHack success rates for GPT-5, used as a controlled testbed for diagnosing failures of primitive control. ***(Right)*** NetHack milestone reach for GPT-5, showing that skill-based control consistently reaches substantially deeper milestones than primitive-only control. Both plots present averages over 32 episodes per task.

show that access to higher-level skills makes `GPT-5` substantially more reliable at tasks requiring navigation, item use, combat, and combinations of these behaviors, although `Quest-Hard` remains far from consistently solved.

RF2: Skills improve NetHack performance across model families. In the 14-model zero-shot sweep, skill-only agents achieve 2.9x the progression, 3.8x the score, and 2.4x the dungeon depth of primitive-only agents on average (Table 1). Measured by dungeon depth, the relative advantage of skills generally increases with model size. This trend is clearest for Llama, where the skill-to-primitive depth ratio increases from 1.7x at 3B to 3.0x at 70B (Table 4).

To make these averages more concrete, Fig. 3 shows milestone reach for `GPT-5`, the strongest model in our zero-shot sweep. The fraction of episodes reaching successive dungeon levels and landmarks such as the Oracle and Gnomish Mines shows how often the agent advances beyond the opening levels and how far it reaches into the dungeon (Klissarov et al., 2025b). With skills, `GPT-5` reaches dungeon level 4 in 84% of episodes, compared with 3% under primitive control, and the Gnomish Mines in 47% versus 3%. These are substantial early-game gains, but even `GPT-5` with skills remains far from solving NetHack (see Discussion). Per-model results and milestone reach rates averaged across models are provided in Tables 4 and 5.

RF3: Skills improve the performance-cost frontier. Fig. 4 plots dungeon depth against estimated inference cost per episode, using OpenRouter input and output token prices[1]. We observe two patterns. First, with skill-only or mixed control, models with higher per-episode costs generally reach deeper into the dungeon, whereas primitive-only performance remains largely flat across costs. Second, for most models, switching from primitives to skills moves the corresponding point up and to the left: the same model reaches deeper dungeon levels at a lower inference cost per episode.

Table 1 quantifies these savings: averaged across models, skills reduce inference cost per episode by 86% and token usage by 74% relative to primitives (see Table 4 for the per-model cost breakdown). Skills lower inference cost by reducing the number of LM calls per episode by an average factor of 5.1. This reduction happens because many environment steps are taken by a code skill policy, which can run quickly and cheaply on the CPU, instead of the more expensive language model policy.

RF4: Mixed control mostly preserves the benefits of abstraction while reducing dependence on library coverage. Mixed control retains 95% of skill-only progression, 91% of score, and 97% of dungeon depth, remaining much stronger than primitives alone (Table 1). It occasionally exceeds

[1] `https://openrouter.ai`

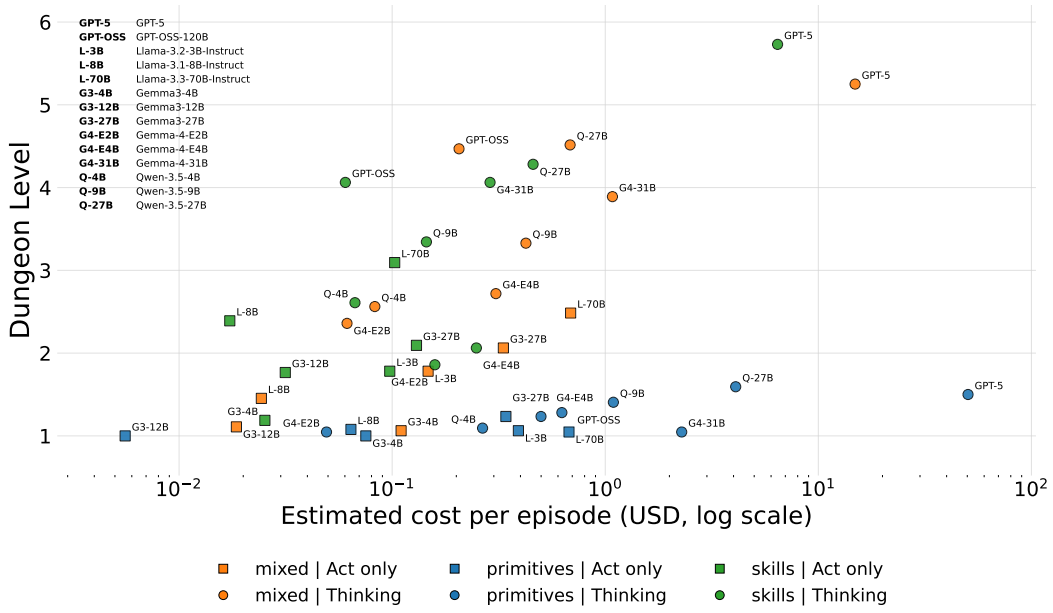

Figure 4: **Zero-shot NetHack cost-performance frontier.** Each point represents a model evaluated with a particular action interface, showing its average dungeon level reached against its average inference cost per episode. For most models, switching from primitives to skills increases dungeon depth while reducing inference cost.

skill-only performance, but raises average inference cost to 2.3x and token usage to 2x that of skills. Access to primitives lets the controller attempt behaviors that skills cannot provide, while requiring it to decide both what to do and which abstraction level to use.

To examine whether primitives help compensate for incomplete skill coverage, we remove one skill family at a time and compare zero-shot `Gemma-4-31B` under skill-only and mixed control (Appendix C.1). Removing skill families can reduce performance in both settings, but mixed control is generally less affected. In the skill-only setting, losing essential skills can severely restrict further progress, whereas mixed control can fall back on primitives to continue advancing through the dungeon.

Two selected zero-shot `GPT-5` trajectories illustrate how primitive fallback works in practice (Appendix C.2). In a vault, the library lacks a skill for dropping gold to satisfy the guard's demand. After several unproductive calls, the agent uses primitives to drop the gold and move along the exit corridor, then resumes navigation with `explore`. In the ranged-combat example, `fight_ranged` returns without acting; the agent uses primitives to throw a javelin before using `pickup_weapon` to recover its weapons. In both cases, primitives resolve a local problem that the available skills cannot handle, allowing the agent to return to skill-based control.

RF5: RL amplifies the performance gap between skill-based control and primitive-only control. We report the evaluation results for the final checkpoint performance of RL in Table 2. We find that both skill-only and mixed control exhibit faster learning than primitive-only control, as demonstrated by a larger performance gain using the same training budget. For example, the gain in dungeon level averaged across models for skill-only and mixed control is 7.2x and 8.6x larger than that for primitive-only control, respectively. We also provide learning curves in Figs. 18 to 21 of Appendix D, which provide further evidence that skill and mixed agents improve faster than primitive agents throughout training, both as a function of gradient steps and wall-clock time. The immediate consequence of faster learning for skills and mixed is that the performance gap between these interfaces and the primitive-only interface is larger at the end of RL training than at the start, validating our finding that RL *amplifies* the performance gap.

RF6: SFT initialization improves final skill-only RL performance. To test whether teacher demonstrations can further improve skill-based NetHack control, we combine SFT with RL, following prior work (Wolczyk et al., 2024; Silver et al., 2016; Guo et al., 2025). At the same RL checkpoint,

Table 2: **Results across learning regimes for `Llama-3.1-8B-Instruct` and `Qwen-3.5-4B`.** Both models use act-only control. We report final score, progression, and dungeon level reached. Evaluation checkpoints are picked at 264 gradient steps for all RL results. Values are mean ± 1 standard error (SE). For zero-shot and SFT, SE is computed across 64 evaluation episodes, while for RL (Base) and RL (SFT) it is computed across 3 training seeds (each seed averages over its own 64 evaluation episodes).

Regime	Interface	Model	Score	Progression	Dungeon Level
Zero-shot	Primitives	`Llama-3.1-8B-Instruct`	22.0 ± 2.9	0.12 ± 0.05	1.08 ± 0.03
	Primitives	`Qwen-3.5-4B`	36.3 ± 6.5	0.29 ± 0.08	1.17 ± 0.06
	Mixed	`Llama-3.1-8B-Instruct`	78.6 ± 8.0	0.90 ± 0.11	1.48 ± 0.08
	Mixed	`Qwen-3.5-4B`	92.0 ± 11.1	0.94 ± 0.11	1.81 ± 0.12
	Skills	`Llama-3.1-8B-Instruct`	**178.4** ± 15.5	**1.59** ± 0.10	**2.48** ± 0.15
	Skills	`Qwen-3.5-4B`	**215.9** ± 22.1	**1.49** ± 0.13	**2.45** ± 0.18
RL (Base)	Primitives	`Llama-3.1-8B-Instruct`	203.1 ± 40.7	1.54 ± 0.23	1.46 ± 0.14
	Primitives	`Qwen-3.5-4B`	334.9 ± 42.6	1.94 ± 0.01	1.54 ± 0.03
	Mixed	`Llama-3.1-8B-Instruct`	477.0 ± 44.5	3.37 ± 0.02	4.80 ± 0.12
	Mixed	`Qwen-3.5-4B`	612.6 ± 32.1	3.64 ± 0.15	4.97 ± 0.14
	Skills	`Llama-3.1-8B-Instruct`	**549.5** ± 40.7	**3.45** ± 0.38	**4.85** ± 0.26
	Skills	`Qwen-3.5-4B`	**674.5** ± 49.9	**4.36** ± 0.24	**5.51** ± 0.20
SFT	Skills	`Llama-3.1-8B-Instruct`	542.4 ± 55.2	3.49 ± 0.31	4.58 ± 0.31
	Skills	`Qwen-3.5-4B`	625.8 ± 54.7	3.41 ± 0.25	4.70 ± 0.27
RL (SFT)	Skills	`Llama-3.1-8B-Instruct`	831.0 ± 67.1	4.46 ± 0.09	5.52 ± 0.18
	Skills	`Qwen-3.5-4B`	1015.7 ± 93.1	5.25 ± 0.49	5.86 ± 0.39

SFT initialization yields higher mean performance than initialization from the original models (Table 2). Mean progression increases from 3.45% to 4.46% for `Llama-3.1-8B-Instruct` and from 4.36% to 5.25% for `Qwen-3.5-4B`, relative improvements of 29% and 20%, respectively. Mean scores increase by approximately 51% for both models. Although the SFT-initialized policies gain less during RL, their stronger starting points translate into better final policies within the evaluated budget. Teacher imitation therefore provides a useful initialization for learning skill-based control, with the additional cost of collecting demonstrations and supervised training.

6 Related work

Language models for low-level control. Prior work has studied entry points for language models into standard reinforcement learning and control frameworks, including using them as reward functions (Ma et al., 2024; Klissarov et al., 2024), for writing code for low-level policies (Kwon et al., 2023; Huang et al., 2023), and (most relevant to our discussion) as a high-level policy that sequences skills (Ahn et al., 2022; Wang et al., 2024; Shentu et al., 2024). In robotics, Liang et al. (2023) use a language model to recursively generate code defining task-specific low-level policies that achieve specified goals. Our work similarly uses code to define policies, but differs by having the LLM call upon an existing set of code policies and sequence them to perform long-horizon tasks. Our skill-based controller is reminiscent of (Ahn et al., 2022), which likewise uses the language model directly as the controller over skills; the aim of our work is different in two ways: (1) to study the value of those skills as compared to primitives and (2) to understand the impact of finetuning the LLM through SFT and RL. In games, Klissarov et al. (2025b) focus on learning parametric skills using AI feedback and composing them through code defining the high-level policy, whereas our work instead uses code as low-level skills and leverages a possibly fine-tuned LLM as a high-level policy. Wang et al. (2024) also use the language model to call code functions that act as skills, but do not study the value of those skills as compared to primitives and instead focus on updating the set of skills for continual progress in an open-ended environment. Several prior works use NetHack as a rich testbed for evaluating language agents (Paglieri et al., 2025; Jeurissen et al., 2024; Piterbarg et al., 2024). Amongst these, our work is most similar to Jeurissen et al. (2024), which likewise uses code-based skills to play NetHack with LLMs. However, they do not provide a systematic comparison between primitive and skill-based play. The `CodeHack` runtime developed in this paper is designed to run with both code skills and low-level primitives, allowing us to carefully ablate the choice of action interface while keeping everything else fixed.

Hierarchical RL. Foundational work on temporal abstractions in reinforcement learning goes back to feudal RL (Dayan & Hinton, 1992), the hierarchy of abstract machines (Parr & Russell, 1997), and options (Sutton et al., 1999). The latter has spurred rich follow-up work, including studies of option discovery and learning (McGovern & Barto, 2001; Stolle & Precup, 2002; Konidaris & Barto, 2009), as well as methods that aim to simultaneously learn the high-level controller along with the options for low-level control (Bacon et al., 2017; Nachum et al., 2018; Khetarpal et al., 2020; Colas et al., 2020; Wulfmeier et al., 2021). A related line of work aims to learn low-level controllers without access to a reward function (i.e., in an unsupervised fashion) (Machado et al., 2017; Eysenbach et al., 2018; Hansen et al., 2019; Sharma et al., 2019; Park et al., 2024). While our work is conceptually motivated by the arguments for hierarchy put forth in this prior work, it is structurally most similar to Parr & Russell (1997) in that we assume skills are given as input as a form of prior knowledge. By making this assumption, we can precisely quantify how much a strong repertoire of skills can accelerate learning, abstracting away (the important and still open) problem of how such skills might be learned. As Sutton et al. (1999) introduce the options framework, they also show that learning and planning with predefined options can drastically improve sample efficiency. We take inspiration from their findings and add to the understanding of the trade-offs of learning with an existing options set in the context of open-ended environments by leveraging large language models. In addition, there have also been several related works that, similar to our work, study the benefits of hierarchy. Park et al. (2023) highlight that hierarchy helps improve the signal-to-noise ratio, which helps with planning. Nachum et al. (2019) find that most of the HRL benefits come from improved exploration. Frans et al. (2018) define a skill discovery objective that directly optimizes for transferability across tasks. Perhaps most relatedly, Piterbarg et al. (2023) finds hierarchy to help in NetHack, but unlike our work focuses mostly on the imitation learning setting and does not use language models. For an extensive survey of HRL, we refer the reader to Klissarov et al. (2025a).

7 Discussion

By providing a systematic comparison across skill, mixed, and primitive control, our work suggests that the action interface can strongly influence the performance of language agents in long-horizon environments requiring low-level control. At the same time, the current results that use `CodeHack` do not come close to solving NetHack. The skill library is hand-designed and necessarily incomplete, so its benefits depend on whether the provided abstractions match the situations encountered by the agent. Some failure modes include situations requiring fine-grained tactical control, rare game knowledge, long-term resource management, or strategic planning beyond what the current skills encode. To deal with these failures, future language agents will need to be able to proficiently move up and down the abstraction ladder by knowing when to use higher-level skills vs. when it's more appropriate to use the low-level primitives (see Appendix C.2 for examples). We believe our finding that mixed control can already retain most of the benefits of skill-only control is a promising result along this path.

A promising direction for future work is for LLMs to construct both action skills and tools for querying remembered information. Agents could identify repeated failures, implement missing action skills, and test and refine them through interaction (Liang et al., 2023; Wang et al., 2024). Alongside these skills, they could construct tools that track previously observed entities and retrieve relevant information on demand, reducing the need to repeatedly process a growing observation history. We see this combination of skill construction, selective information access, and primitive fallback as a plausible path toward solving NetHack, allowing agents to both move up and down the abstraction ladder and to extend this ladder through experience.

AI Use Statement

We used language models to assist with coding, drafting portions of the manuscript and polishing the text for clarity and readability. We also used a `Gemma-4-31B` model to produce 1,024 zero-shot NetHack teacher trajectories for supervised fine-tuning, as described in Section 4 and Appendix B.4. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

Cédric Colas, Tristan Karch, Nicolas Lair, Jean-Michel Dussoux, Clément Moulin-Frier, Peter Ford Dominey, and Pierre-Yves Oudeyer. Language as a Cognitive Tool to Imagine Goals in Curiosity-Driven Exploration. *Neural Information Processing Systems*, 2020.

Peter Dayan and Geoffrey E Hinton. Feudal reinforcement learning. In S. Hanson, J. Cowan, and C. Giles (eds.), *Advances in Neural Information Processing Systems*, volume 5. Morgan-Kaufmann, 1992. URL `https://proceedings.neurips.cc/paper_files/paper/1992/file/d14220ee66aeec73c49038385428ec4c-Paper.pdf`.

Aniket Didolkar, Anirudh Goyal, Nan R Ke, Siyuan Guo, Michal Valko, Timothy Lillicrap, Danilo Rezende, Yoshua Bengio, Michael Mozer, and Sanjeev Arora. Metacognitive capabilities of llms: An exploration in mathematical problem solving. *Advances in Neural Information Processing Systems*, 37:19783–19812, 2024.

Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. In *International Conference on Learning Representations*, 2018.

Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta Learning Shared Hierarchies. In *International Conference on Learning Representations*, 2018.

Team Gemma. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

Nikolaj Goodger, Peter Vamplew, Cameron Foale, and Richard Dazeley. A nethack learning environment language wrapper for autonomous agents. *Journal of Open Research Software*, 11, 2023. doi: 10.5334/jors.444.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Tingxu Han, Yi Zhang, Wei Song, Chunrong Fang, Zhenyu Chen, Youcheng Sun, and Lijie Hu. Swe-skills-bench: Do agent skills actually help in real-world software engineering? *arXiv preprint arXiv:2603.15401*, 2026.

Steven Hansen, Will Dabney, Andre Barreto, David Warde-Farley, Tom Van de Wiele, and Volodymyr Mnih. Fast task inference with variational intrinsic successor features. In *International Conference on Learning Representations*, 2019.

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. Voxposer: Composable 3d value maps for robotic manipulation with language models. *arXiv preprint arXiv:2307.05973*, 2023.

Dominik Jeurissen, Diego Perez-Liebana, Jeremy Gow, Duygu Cakmak, and James Kwan. Playing nethack with llms: Potential & limitations as zero-shot agents. In *2024 IEEE Conference on Games (CoG)*, pp. 1–8. IEEE, 2024.

Khimya Khetarpal, Martin Klissarov, Maxime Chevalier-Boisvert, Pierre-Luc Bacon, and Doina Precup. Options of interest: Temporal abstraction with interest functions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 4444–4451, 2020.

Martin Klissarov, Pierluca D'Oro, Shagun Sodhani, Roberta Raileanu, Pierre-Luc Bacon, Pascal Vincent, Amy Zhang, and Mikael Henaff. Motif: Intrinsic Motivation from Artificial Intelligence Feedback. *International Conference on Learning Representations*, 2024.

Martin Klissarov, Akhil Bagaria, Ziyan Luo, George Konidaris, Doina Precup, and Marlos C Machado. Discovering temporal structure: An overview of hierarchical reinforcement learning. *arXiv preprint arXiv:2506.14045*, 2025a.

Martin Klissarov, Mikael Henaff, Roberta Raileanu, Shagun Sodhani, Pascal Vincent, Amy Zhang, Pierre-Luc Bacon, Doina Precup, Marlos C Machado, and Pierluca D'Oro. Maestromotif: Skill design from artificial intelligence feedback. In *The Thirteenth International Conference on Learning Representations*, 2025b.

George Dimitri Konidaris and Andrew G. Barto. Skill Discovery in Continuous Reinforcement Learning Domains using Skill Chaining. In *Neural Information Processing Systems*, 2009.

Heinrich Kuttler, Nantas Nardelli, Alexander H. Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktaschel. The nethack learning environment. In *Advances in Neural Information Processing Systems*, 2020.

Teyun Kwon, Norman Di Palo, and Edward Johns. Language Models as Zero-Shot Trajectory Generators. In *IEEE Robotics and Automation Letters*, 2023.

Yingru Li, Jiacai Liu, Jiawei Xu, Yuxuan Tong, Ziniu Li, Qian Liu, and Baoxiang Wang. Trust region masking for long-horizon llm reinforcement learning. *arXiv preprint arXiv:2512.23075*, 2025.

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.

Jiacai Liu, Yingru Li, Yuqian Fu, Jiawei Wang, Qian Liu, and Yu Shen. When speed kills stability: Demystifying RL collapse from the training-inference mismatch, September 2025. URL `https://richardli.xyz/rl-collapse`.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-Level Reward Design via Coding Large Language Models. *International Conference on Learning Representations*, 2024.

Marlos C Machado, Marc G Bellemare, and Michael Bowling. A laplacian framework for option discovery in reinforcement learning. In *International conference on machine learning*, pp. 2295–2304. PMLR, 2017.

Amy McGovern and Andrew G Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *International Conference on Machine Learning*, 2001.

Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. *Advances in neural information processing systems*, 31, 2018.

Ofir Nachum, Haoran Tang, Xingyu Lu, Shixiang Gu, Honglak Lee, and Sergey Levine. Why does hierarchy (sometimes) work so well in reinforcement learning? *arXiv preprint arXiv:1909.10618*, 2019.

Davide Paglieri, Bartlomiej Cupial, Samuel Coward, Ulyana Piterbarg, Maciej Wolczyk, Akbir Khan, Eduardo Pignatelli, Lukasz Kuciski, Lerrel Pinto, Rob Fergus, Jakob Nicolaus Foerster, Jack Parker-Holder, and Tim Rocktaschel. Balrog: Benchmarking agentic llm and vlm reasoning on games. In *International Conference on Learning Representations*, 2025.

Seohong Park, Dibya Ghosh, Benjamin Eysenbach, and Sergey Levine. Hiql: Offline goal-conditioned rl with latent states as actions. *Advances in Neural Information Processing Systems*, 36:34866–34891, 2023.

Seohong Park, Oleh Rybkin, and Sergey Levine. Metra: Scalable unsupervised rl with metric-aware abstraction. In *The Twelfth International Conference on Learning Representations*, 2024.

Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, 10, 1997.

Ulyana Piterbarg, Lerrel Pinto, and Rob Fergus. Nethack is hard to hack. *Advances in Neural Information Processing Systems*, 36:37540–37566, 2023.

Ulyana Piterbarg, Lerrel Pinto, and Rob Fergus. diff history for neural language agents. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 40843–40868, 2024.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Mikayel Samvelyan, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Kuttler, Edward Grefenstette, and Tim Rocktaschel. Minihack the planet: A sandbox for open-ended reinforcement learning research. *arXiv preprint arXiv:2109.13202*, 2021.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017. URL https://api.semanticscholar.org/CorpusID:28695052.

Archit Sharma, Shixiang Gu, Sergey Levine, Vikash Kumar, and Karol Hausman. Dynamics-aware unsupervised discovery of skills. In *International Conference on Learning Representations*, 2019.

Yide Shentu, Philipp Wu, Aravind Rajeswaran, and Pieter Abbeel. From llms to actions: Latent codes as bridges in hierarchical robot control. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 8539–8546. IEEE, 2024.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi: 10.1038/nature16961. URL https://doi.org/10.1038/nature16961.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation*, pp. 212–223, Berlin, Heidelberg, 2002. Springer-Verlag. ISBN 3540439412.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181–211, 1999. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(99)00052-1. URL https://www.sciencedirect.com/science/article/pii/S0004370299000521.

Jens Tuyls, Dhruv Madeka, Kari Torkkola, Dean Foster, Karthik R Narasimhan, and Sham M. Kakade. Scaling laws for imitation learning in single-agent games. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=qGVchjFj3a.

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.

Maciej Wolczyk, Bartłomiej Cupiał, Mateusz Ostaszewski, Michał Bortkiewicz, Michał Zając, Razvan Pascanu, Łukasz Kuciński, and Piotr Miłoś. Fine-tuning reinforcement learning models is secretly a forgetting mitigation problem. In *International Conference on Machine Learning*, pp. 53039–53078. PMLR, 2024.

Markus Wulfmeier, Dushyant Rao, Roland Hafner, Thomas Lampe, Abbas Abdolmaleki, Tim Hertweck, Michael Neunert, Dhruva Tirumala, Noah Siegel, Nicolas Heess, et al. Data-efficient hindsight off-policy option learning. In *International Conference on Machine Learning*, pp. 11340–11350. PMLR, 2021.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, et al. Osworld2. 0: Benchmarking computer use agents on long-horizon real-world tasks. *arXiv preprint arXiv:2606.29537*, 2026.

A Technical Details of CodeHack

This appendix provides additional implementation detail for `CodeHack`, the code-based skill library used throughout the paper. Whereas Section 3 introduces the library at a high level, the goal of this appendix is to make the artifact easier to understand, reproduce, and extend.

A.1 Library overview

`CodeHack` is implemented as a control layer between the primitive NetHack action interface and higher-level controllers such as language models or reinforcement learning agents. Its role is to expose temporally extended, semantically named actions while preserving a path back to lower-level control when finer intervention is needed.

We developed `CodeHack` iteratively, drawing on human NetHack knowledge to identify reusable behaviors for navigation, exploration, combat, item management, and recovery. We implemented these behaviors as Python procedures and tested them through end-to-end tests, human playthroughs, and `GPT-5` playthroughs. Failures observed during testing guided debugging and refinement of the skill implementations. For the experiments, we prioritized reliable routines that were broadly useful across the evaluated tasks.

Design principles. `CodeHack` is designed around three practical design principles for long-horizon control: **(1) Semantic meaning**: skills should correspond to recognizable behaviors such as exploration, combat, equipment management, or terrain interaction rather than arbitrary low-level command bundles. **(2) Reusability**: skills should be reusable across many states and episodes, so that they function as stable units of abstraction rather than narrow scripts tied to a single layout. **(3) Mobility across abstraction levels**: the interface should support moving up and down the abstraction ladder, so that high-level skills accelerate long-horizon behavior without removing the possibility of more local intervention when abstractions are insufficient. The runtime therefore includes explicit mechanisms for moving back down the abstraction ladder when a high-level skill is not appropriate. Skills can be interrupted by panic handlers when the local state changes unexpectedly, for example when a newly reachable hostile monster appears or the agent loses HP, and the wrapper can report when a selected skill makes no progress. These signals expose abstraction failures early and let the outer controller choose a different high-level behavior or, under mixed control, revert to more local control. The following subsections describe the runtime, state tracking, and execution mechanics.

A.2 WRAPPER AND CONTROL INTERFACE

At runtime, `CodeHack` is introduced by wrapping an NLE or MiniHack environment with a skill-aware wrapper. This wrapper instantiates the internal `CodeHack` runtime, registers a configurable set of strategies, registers panic handlers, and can optionally expose selected primitive commands as one-step strategies. Once wrapped, the outer controller interacts with the environment by choosing from an action space of named skills rather than directly emitting raw NetHack commands.

Skill execution follows a call-and-return discipline. A controller selects one skill, the corresponding Python procedure is invoked, and the procedure continues issuing primitive actions until it returns, triggers a panic, or the episode terminates. After each primitive step, the runtime updates its state so that the skill can respond to new observations before choosing subsequent actions. For example, selecting `goto_room` calls a function that may execute several movement commands based on persistent map memory. Control returns only after the target room is reached, the skill decides that it cannot proceed, an interrupt fires, or the episode ends. The wrapper records how many primitive environment transitions were consumed during the selected skill and exposes this count as `env_steps`; it also aggregates the discounted reward accumulated during those internal steps and stores auxiliary skill-level statistics in the returned info dictionary. An additional feedback wrapper can attach a short text message when a selected skill makes zero primitive steps, giving the outer controller an immediate indication that the chosen abstraction was inapplicable or failed locally.

Because the runtime can register both higher-level strategies and selected primitive commands, the same underlying environment can be exposed as a primitive-only, skill-only, or mixed control interface. This is the main experimental advantage of the wrapper: it lets us vary abstraction level without rewriting the environment or the execution machinery.

A.3 INTERNAL STATE REPRESENTATION

The central runtime abstraction in `CodeHack` is a state manager that maintains a structured, persistent view of the game state on top of raw NLE observations. At every primitive step, this runtime updates its cache, so skills can reason over symbolic entities such as rooms, items, monsters, and reachability instead of repeatedly parsing raw tensors from scratch. The internal state includes the following components:

- **Current observation state**, including the current message, glyph map, terminal characters and colors, cursor position, and `blstats`.
- **Persistent level memory**, stored in a `Level` object that records seen tiles, walkable structure, map objects, visited positions, known traps, door locations, and simple terrain-derived features.
- **Inventory state**, maintained by an `InventoryManager` that parses inventory glyphs and strings into typed items grouped by categories such as armor, rings, potions, food, tools, and wands.
- **Character state**, including role-specific skill information, currently known spells, and equipment-dependent combat properties.
- **Navigation state**, maintained by a pathfinder over currently known walkable positions, with costs adjusted to avoid local hazards such as monsters or traps.
- **Combat and trap utilities**, used by higher-level skills that reason about tactical movement, attack choices, or environmental hazards.

Persistent level memory is especially important for navigation and exploration. A skill can depend not only on what is visible now, but also on previously observed map structure such as earlier rooms, corridors, stairs, doors, and hazards.

A.4 PANIC HANDLERS AND INTERRUPTION

Skill execution is intentionally interruptible. After each primitive step, `CodeHack` runs a configurable list of panic handlers over the updated symbolic state. The current implementation includes handlers

such as `lost_hp`, which aborts when the agent takes damage, and `enemy_appeared`, which aborts when a newly reachable hostile monster enters the local state.

This mechanism reduces the brittleness of longer procedures by preventing them from blindly continuing through clearly adverse local changes. It also creates a clean handoff back to the outer controller: once a panic fires, the controller can choose a different high-level skill or drop to lower-level actions that are better matched to the new local state.

A.5 Experimental skill set

We list here the skills exposed in the main experimental action space, followed by additional skills used in MiniHack-only settings. We group them by broad behavioral role.

Exploration, navigation, and search. `explore`, `goto_corridor`, `goto_corridor_east`, `goto_corridor_north`, `goto_corridor_south`, `goto_corridor_west`, `goto_room`, `goto_room_east`, `goto_room_north`, `goto_room_south`, `goto_room_west`, `descend_stairs`, `ascend_stairs`, `goto_corpse`, `search_corridor_for_hidden_doors`, and `search_room_for_hidden_doors`.

Combat and tactical control. `engrave_elbereth`, `fight_engulfed`, `fight_melee`, `fight_ranged`, `zap_monster`, `approach_monster`, `goto_choke_point`, `wait_for_monster`, and `escape_trap`.

Doors, terrain, and special environment interaction. `open_doors`, `open_doors_kick`, `open_doors_key`, `goto_boulder`, `push_boulder_east`, `push_boulder_north`, `push_boulder_south`, `push_boulder_west`, `leave_shop`, `identify_items_altar`, and `dip_for_excalibur`.

Item acquisition and equipment management. `examine_items`, `pickup_amulet`, `pickup_armor`, `pickup_coin`, `pickup_food`, `pickup_gem`, `pickup_potion`, `pickup_ring`, `pickup_scroll`, `pickup_spellbook`, `pickup_tool`, `pickup_wand`, `pickup_weapon`, `pickup_corpse`, `puton_ring`, `puton_amulet`, `wear_boots`, `wear_cloak`, `wear_gloves`, `wear_helm`, `wear_shield`, `wear_shirt`, and `wear_suit`.

Food, prompts, recovery, and cleanup. `eat_corpse_floor`, `eat_corpse_inventory`, `eat_food_inventory`, `yes`, `no`, `cancel`, `more`, `emergency_escape`, `fix_trouble`, `pray`, `heal`, `rest_until_full_health`, `drop_cursed_items`, `drop_unequipped_armor`, `drop_unequipped_weapons`, `drop_unidentified_potions`, `drop_unidentified_scrolls`, `drop_unidentified_spellbooks`, and `drop_unidentified_wands`.

MiniHack-only skills. `cross_lava_river`, `freeze_lava_horn`, `freeze_lava_river`, `acquire_levitation`, `approach_lava_river`, `levitate_over_lava_river`, and `freeze_lava_wand`

B Additional experimental details

This appendix provides additional details for Section 4. We describe the NetHack and MiniHack environments in more detail, define the progress metric used for NetHack, list the MiniHack tasks used in our study, summarize the prompt templates used by the language agent, and report the main training hyperparameters for SFT and RL.

B.1 Environment details

NetHack. NetHack is a classic and highly complex terminal roguelike game built around dungeon exploration, tactical combat, inventory management, and long-horizon planning. In the full game, the player must descend through more than 50 procedurally generated dungeon levels, retrieve the Amulet of Yendor, and then ascend back to the surface through several extremely difficult late-game

regions. The NetHack Learning Environment (NLE) (Kuttler et al., 2020) wraps the original game in a synchronous RL interface while leaving the underlying game dynamics unchanged. Because the environment is both procedurally generated and stochastic, it is designed to stress exploration, planning, credit assignment, and systematic generalization rather than memorization of fixed layouts.

At the level of the raw simulator, NLE is naturally multimodal. The default observation space includes symbolic views of the visible dungeon map together with textual and structured information such as the current message, the hero's bottom-line statistics, and the inventory. More concretely, the NLE paper describes the default observation as including `glyphs`, `chars`, `colors`, `specials`, `blstats`, `message`, and several inventory arrays (`inv_glyphs`, `inv_strs`, `inv_letters`, and `inv_oclasses`). In our setup, these low-level observations are not passed directly to the language model. Instead, we use the NLE language wrapper to convert them into natural-language descriptions and an ASCII map, producing the text-based interface described in Section 4 and Appendix B.

The original NLE paper defines a full action space of 93 discrete actions, consisting of 77 command actions and 16 movement actions. However, interacting with NetHack through its terminal interface often requires *action chaining*: many semantically simple behaviors are implemented as sequences of low-level key presses. For example, commands such as throwing, zapping, or applying an item typically require an initial command followed by one or more menu selections, inventory letters, prompt confirmations, or directions. This substantially increases the difficulty of primitive control because an agent must not only decide *what* to do, but also produce the correct sequence of interface-level actions needed to complete the command. In our code-wrapper setup, this is reflected in the primitive language interface, which exposes an expanded set of textual action strings including command names, directional movement, prompt responses, and menu-selection characters.

Our main NetHack training environment is `NetHackScore-v0`. In this task, the environment reward is the change in the in-game NetHack score between successive timesteps. As emphasized by the NLE paper, this score is a sensible proxy for incremental progress because it depends on factors such as dungeon depth, enemy kills, gold collection, and object knowledge. At the same time, score is not perfectly aligned with actually solving the game: expert human players can ascend while deliberately keeping score relatively low. For this reason, in addition to score we also report the BALROG NLE-progress metric described below.

Procedural generation also matters for the interpretation of our results. Since each episode is generated from a fresh random seed, the agent is highly unlikely to encounter exactly the same level twice. The goal is therefore not to memorize solutions to specific layouts, but to learn reusable behavioral patterns that transfer across unseen seeds. Moreover, progression in NetHack is not strictly linear: the agent may backtrack, revisit earlier levels, or enter branching sub-dungeons such as the Gnomish Mines. This means that even within a single episode, the agent must preserve and reuse knowledge across changing contexts rather than merely advance through a fixed curriculum of stages. In our experiments, NetHack serves as the main long-horizon domain of study, and our SFT and RL experiments focus on this setting.

Unless noted otherwise, NetHack runs use the `val-hum-law-fem` character, disable autopickup, keep the pet enabled, allow unrestricted menu and `yn` responses, skip repeated `-More-` prompts automatically, and use a 5,000-step episode limit with a 150-step no-progress timeout.

NetHack progress metric. For NetHack, in addition to raw environment score, we report the NLE-progress metric introduced in BALROG (Paglieri et al., 2025). The motivation is that in-game NetHack score does not always align with meaningful progress toward winning the game. BALROG therefore defines a data-driven progression metric from human-played NetHack trajectories. Let $d^\star$ denote the highest dungeon level reached in an episode and $x^\star$ the highest experience level reached. BALROG maps each of these to a probability of eventual human ascension using empirical progression curves, with Dungeon Level 1 and Experience Level 1 defined as 0% progress and ascension defined as 100%. The episode-level progress score is then

$$\text{progress} = \max\big(p_{\text{DLvl}}(d^\star),\ p_{\text{xlvl}}(x^\star)\big),$$

where p_{DLvl} and p_{xlvl} are the BALROG progression curves. We use this metric as our main measure of human-like game progress in NetHack.

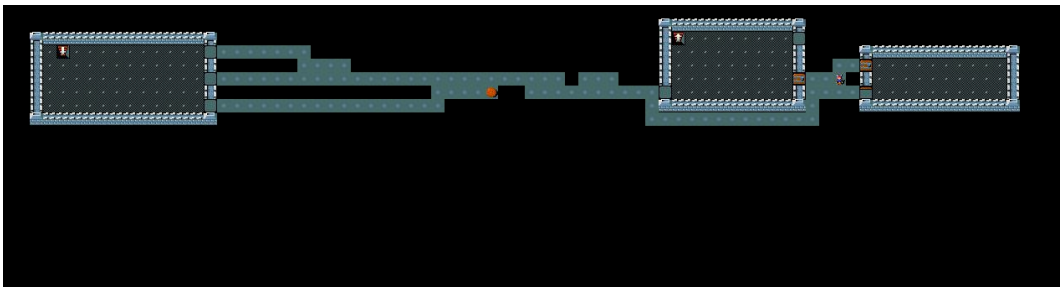

Figure 5: Example layout from `MiniHack-Corridor-R3-v0`.

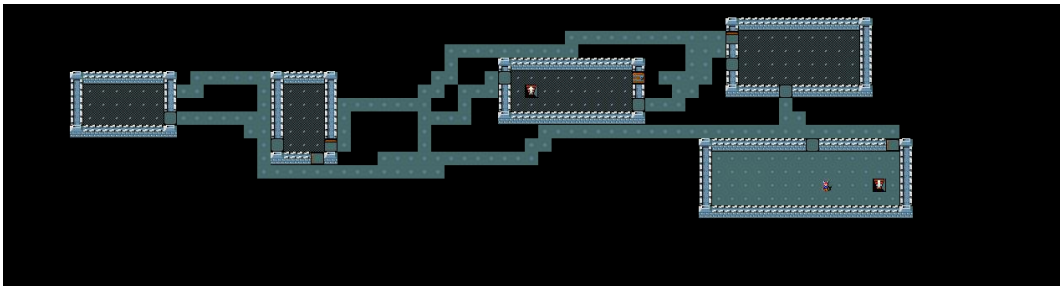

Figure 6: Example layout from `MiniHack-Corridor-R5-v0`

MiniHack. MiniHack (Samvelyan et al., 2021) is a flexible sandbox built on top of NLE that allows researchers to define controlled NetHack-style tasks with custom layouts, monsters, terrain, and objects. Following BALROG (Paglieri et al., 2025), we use MiniHack as a complementary zero-shot testbed because it preserves the mechanics of NetHack while making failures easier to interpret. To interface with language models, we use the same NLE language wrapper as in NetHack, so both domains are presented through a shared text-based observation and action interface.

B.2 MiniHack task suite

Our MiniHack evaluation suite covers seven tasks spanning exploration, combat, inventory usage, and long-horizon subgoal sequencing. The standard Corridor and Quest families come directly from MiniHack, while our Corridor-R10 evaluation level is a custom extension of the Corridor family and simulates exploration of a single NetHack level.

Corridor-R3 (`MiniHack-Corridor-R3-v0`). Corridor-R3 belongs to the MiniHack Corridor family of exploration tasks. The objective is to reach the staircase located in one of the rooms, but the room positions, room sizes, and connecting corridors are procedurally generated. In the R3 version, the level is composed of three rooms. To solve it reliably, the agent has to navigate rooms and corridors, open doors, including kicking closed ones when needed, and search for hidden doors and hidden corridors that may block progress. This creates a short-horizon exploration problem in which the agent must execute the core mechanics of NetHack navigation in a relatively small layout. Corridor-R3 is useful for exposing brittle action repetition and basic exploration failures under primitive control. Fig. 5 shows a representative state.

Corridor-R5 (`MiniHack-Corridor-R5-v0`). Corridor-R5 uses the same basic construction as Corridor-R3, but increases the number of rooms to five. This directly increases the amount of exploration needed before the staircase can be found, and it raises the cost of wasted movement or local oscillation. Compared with R3, the task places more pressure on maintaining directional progress, remembering which parts of the map have already been explored, and recovering from short detours without becoming trapped in repetitive low-level behaviors. Fig. 6 shows a representative state.

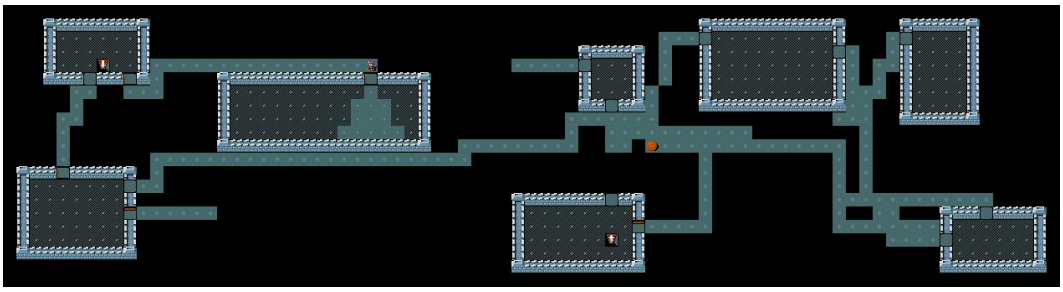

Figure 7: Example layout from `MiniHack-Corridor-R10-v0`

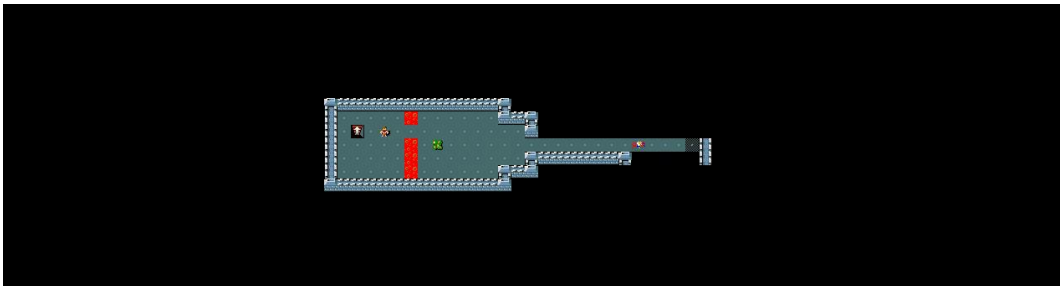

Figure 8: Example layout for `MiniHack-Quest-Easy-v0`.

Corridor-R10 (`MiniHack-Corridor-R10-v0`). Corridor-R10 is a custom extension of the standard Corridor family that further increases the number of rooms. This task is particularly useful because it tests whether an agent can explore a normal NetHack-style corridor layout while disentangling exploration from survival. The challenge is not to win difficult fights or manage complex item interactions, but to make steady exploratory progress over a longer navigation horizon. Fig. 7 shows a representative state.

Quest-Easy (`MiniHack-Quest-Easy-v0`). This task is the simplest member of the Quest family. The layout is relatively simple and mostly fixed, so the main difficulty is not large-scale search but correctly sequencing a short multi-stage plan. The agent must explore the map, acquire or use an object that allows it to cross a lava river, survive a small amount of combat, and then reach the staircase. Compared with pure navigation tasks, Quest-Easy introduces basic item usage and simple combat. Fig. 8 shows a representative state.

Quest-Medium (`MiniHack-Quest-Medium-v0`). Quest-Medium increases the difficulty of combat and subgoal coordination. In this variant, the agent must fight a swarm of monsters in a narrow corridor, where careless forward movement can easily lead to being surrounded. At the same time, the environment tests whether the agent can recognize that an item may be useful for a later subgoal rather than immediately: after surviving the encounter with monsters, it must use the item found at the beginning of the level, and use it to cross the lava river. Only then can it reach the staircase. Successful behavior therefore requires both corridor-aware combat and delayed item use. Fig. 9 shows a representative state.

Quest-Hard (`MiniHack-Quest-Hard-v0`). Quest-Hard is the most demanding task in this family. It begins with a large procedurally generated maze that must be solved before the rest of the quest can even begin. If the agent spends too much time exploring it will die because of hunger forcing the agent to be efficient. After escaping the maze, the agent still needs to collect and use items to cross the lava river, and finally use a wand of death to kill a minotaur guarding the staircase. Overall, Quest-Hard is a long-horizon problem that requires chaining together multiple skills, including efficient exploration, item collection, lava crossing, and combat. Fig. 10 shows a representative state.

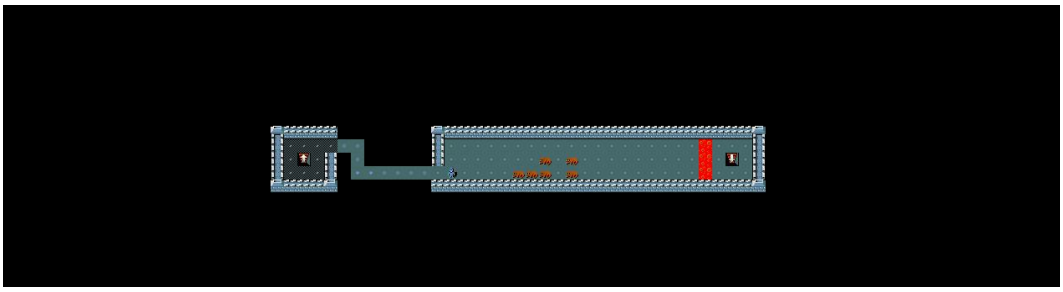

Figure 9: Example layout for `MiniHack-Quest-Medium-v0`.

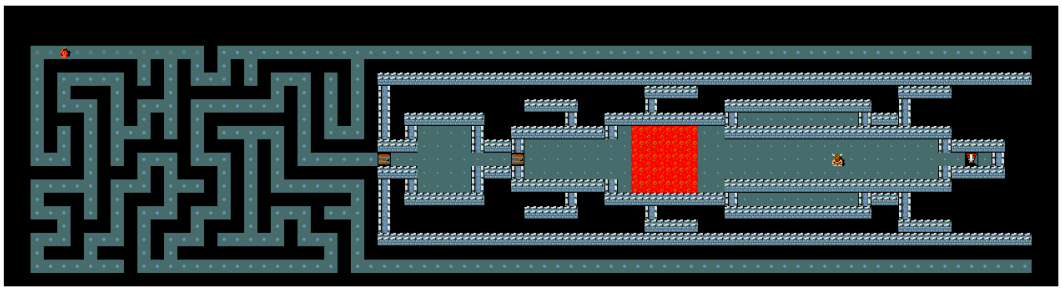

Figure 10: Example layout for `MiniHack-Quest-Hard-v0`.

WoD-Hard-Full (`MiniHack-WoD-Hard-Full-v0`). This task belongs to the Wand-of-Death family. The central mechanic is that the agent must use a wand of death correctly to eliminate a dangerous monster that blocks progress to the staircase. In the hard variant, the wand must first be found or acquired before it can be used, and used to kill the minotaur. As a result, the task combines item pickup, item use, and combat. This is useful because correct wand-of-death usage is itself an important subtask inside `MiniHack-Wod-Hard-Full-v0`. Fig. 11 shows a representative state.

B.3 Prompt templates and evaluation protocol

For each task, we use the same observation format and prompt template across *primitive*, *skill*, and *mixed* settings. The available-action lists and descriptions change to match the selected interface.

System prompt. The system prompt has the following structure:

```
You are an expert NetHack player. Your task is to: {task_goal}

<task_instruction>
- Analyze the observation.
- Choose the best immediate action from the command/tool list.
- Provide ONLY the action command inside <action> tags.
</task_instruction>

<commands>...</commands>    (primitive or mixed settings)
<tools>...</tools>          (skill or mixed settings)
<tips>...</tips>
<output_format><action>...</action></output_format>
```

In the primitive setting, `<commands>` contains the low-level action list. In the skill setting, `<tools>` contains the available `CodeHack` skills together with short docstring-based descriptions. In the mixed setting, both blocks are present.

User turn structure. At each timestep t, the model receives a chat-style user message containing recent action history and the current text observation:

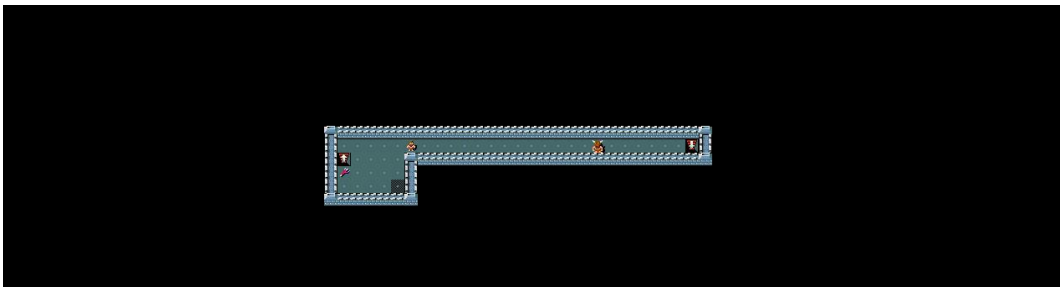

Figure 11: Example layout for `MiniHack-WoD-Hard-Full-v0`.

```
<action_history>...</action_history>
<current_state>
  <message_log>...</message_log>
  <cursor>...</cursor>
  <map description>...</map description>
  <stats>...</stats>
  <map>...</map>
  <language_observation>...</language_observation>
  <inventory>...</inventory>
  <feedback>...</feedback> (optional)
</current_state>
```

The `<feedback>` block is used to tell the agent when its previous output was malformed or mapped to an invalid action. The model must answer with a single action string enclosed in `<action>` tags.

History and fallback. In the main setup used for zero-shot evaluation, SFT, and RL, the agent keeps five interaction turns of history and is given the two most recent observations. If the model output does not contain a valid `<action>` block, the environment replaces it with a default action and tells the agent that its previous action was invalid. This fallback mechanism prevents brittle formatting failures from terminating evaluation episodes.

B.4 Training details

Table 3 summarizes the main hyperparameters used in SFT and RL. Unless noted otherwise, the reported SFT and RL experiments use a 5,000-step episode limit and set the code-wrapper's maximum skill horizon to 100,000 primitive steps.

During SFT, we train a LoRA adapter on top of the base language model. In the current setup, this uses rank-stabilized LoRA with rank 128 and scaling 64. After SFT, the learned adapter is merged back into the base model. This merged model is then used as the initialization for the SFT-primed RL experiments described below.

For RL from SFT, the model is initialized from the SFT-primed model obtained by merging the SFT adapter into the base model. For RL from base, the model is trained directly from the base model without SFT warm-starting. The default RL configuration uses rank-stabilized LoRA with rank 128 and scaling 64, 32 rollouts per trainer GPU, 16 controller decisions per rollout, 350 gradient steps, $\lambda = 0.95$, clip range 0.2, and policy and value learning rates of 2×10^{-6}.

We use Ray to orchestrate data collection and training, vLLM for inference, and PyTorch for training. Each environment worker maintains a separate environment instance and sends its observation and interaction history to vLLM to generate the next action. We collect fixed-length rollouts, storing the prompt, generated tokens, reward, termination flags, and the sequence log-probability returned by vLLM. Once a batch is complete, the trainer computes value estimates and generalized advantage estimates, then updates the policy and value models using PPO minibatches.

We use double buffering to overlap data collection with training: while the trainer updates on one batch, the environment workers collect the next. This reduces idle time but introduces policy lag, as

Table 3: Main SFT and RL hyperparameters used in our experiments.

Stage	Hyperparameter	Value
SFT	Teacher model	Gemma-4-31B
SFT	Teacher episodes	1,024 zero-shot episodes
SFT	Trainable parameters	LoRA adapter only
SFT	LoRA rank / alpha / dropout	128 / 64 / 0.0
SFT	Learning rate	5×10^{-5}
SFT	Local minibatch size	64
SFT	Per-device train batch size	2
SFT	Training epochs	1
SFT	Evaluation frequency	every 8,192 samples
SFT	Max generated action tokens	32 in the non-thinking setup
RL	Algorithm	PPO
RL	Initializations compared	base model and merged SFT model
RL	Trainable parameters	LoRA adapters for both policy and value
RL	LoRA rank / alpha / dropout	128 / 64 / 0.0
RL	Policy / value learning rates	2×10^{-6} / 2×10^{-6}
RL	KL coefficient	0.0
RL	Rollouts per trainer GPU	32
RL	Controller decisions per rollout	16
RL	Local minibatch size	256
RL	Per-device train batch size	8
RL	Gradient steps	350
RL	GAE λ	0.95
RL	PPO clip range	0.2
RL	Entropy coefficient	0.0
RL	Number of seeds	3

the collected data may come from an earlier version of the policy. Numerical differences between vLLM and the PyTorch training backend can also produce different action probabilities, even with the same model weights. Such training inference mismatch has been identified as a source of instability (Liu et al., 2025; Li et al., 2025). To account for both sources of off-policy mismatch, we compute the PPO importance ratio using the log-probabilities recorded by vLLM during collection.

Each reinforcement learning and supervised fine-tuning run was completed in under one day using 8xH100 GPUs, while each zero-shot and evaluation run was completed in under one day using a single H100 GPU.

C FULL ZERO-SHOT NETHACK RESULTS

For completeness, Table 4 reports the full aggregate zero-shot NetHack results for every model and action interface used in our sweep. We use reasoning mode for GPT-5, GPT-OSS-120B, Gemma-4, and Qwen-3.5, and act-only mode for Llama and Gemma3.

Table 4: Aggregate NetHack zero-shot results for the same 14 models and thinking modes as Table 1. Gray ± terms are standard errors across episodes. Progression is in percent; milestone reach values are fractions. All settings use 64 evaluation episodes except `GPT-5`, which uses 32 episodes per action interface due to cost. Mines reach is computed from episode traces.

Model	Action	Progression	Score	Dlvl	L2	L4	L6	L8	Oracle	Mines	Cost/ep. ($)	Tokens/ep.
GPT-5	skills	4.64 ± 0.38	1279.29 ± 151.62	5.73 ± 0.29	0.94 ± 0.03	0.84 ± 0.05	0.54 ± 0.06	0.25 ± 0.06	0.06 ± 0.03	0.47 ± 0.06	6.44	2,007,700
	mixed	3.99 ± 0.54	922.84 ± 179.63	5.25 ± 0.42	0.94 ± 0.04	0.78 ± 0.07	0.38 ± 0.09	0.22 ± 0.07	0.06 ± 0.04	0.53 ± 0.09	14.88	5,196,745
	primitives	1.78 ± 0.13	232.56 ± 22.59	1.50 ± 0.14	0.34 ± 0.09	0.03 ± 0.03	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.03 ± 0.03	50.41	13,866,723
GPT-OSS-120B	skills	2.88 ± 0.17	447.72 ± 39.81	4.06 ± 0.23	0.95 ± 0.03	0.61 ± 0.06	0.22 ± 0.05	0.05 ± 0.03	0.03 ± 0.02	0.45 ± 0.06	0.06	1,176,438
	mixed	3.17 ± 0.22	495.48 ± 41.31	4.47 ± 0.25	0.95 ± 0.03	0.67 ± 0.06	0.33 ± 0.06	0.06 ± 0.03	0.05 ± 0.03	0.36 ± 0.06	0.21	4,151,999
	primitives	1.39 ± 0.11	152.97 ± 14.46	1.28 ± 0.08	0.20 ± 0.05	0.02 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.63	10,399,676
Llama-3.2-3B-Instruct	skills	0.88 ± 0.11	53.78 ± 7.03	1.78 ± 0.11	0.52 ± 0.06	0.03 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.06 ± 0.03	0.10	1,870,996
	mixed	1.02 ± 0.10	62.09 ± 6.90	1.78 ± 0.10	0.62 ± 0.06	0.02 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.08 ± 0.03	0.15	2,842,893
	primitives	0.10 ± 0.05	15.94 ± 2.83	1.06 ± 0.03	0.06 ± 0.03	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.02 ± 0.02	0.39	7,495,379
Llama-3.1-8B-Instruct	skills	1.59 ± 0.10	178.39 ± 15.52	2.48 ± 0.15	0.81 ± 0.05	0.11 ± 0.04	0.03 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.17 ± 0.05	0.02	858,096
	mixed	0.90 ± 0.11	78.62 ± 7.98	1.48 ± 0.08	0.42 ± 0.06	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.06 ± 0.03	0.02	1,210,851
	primitives	0.12 ± 0.05	21.98 ± 2.91	1.08 ± 0.03	0.08 ± 0.03	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.06	3,181,590
Llama-3.3-70B-Instruct	skills	2.24 ± 0.17	332.19 ± 37.04	3.09 ± 0.20	0.83 ± 0.05	0.39 ± 0.06	0.09 ± 0.04	0.00 ± 0.00	0.02 ± 0.02	0.28 ± 0.06	0.10	1,020,211
	mixed	1.79 ± 0.17	334.45 ± 40.69	2.48 ± 0.18	0.66 ± 0.06	0.28 ± 0.06	0.03 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.27 ± 0.06	0.69	6,842,115
	primitives	0.76 ± 0.12	79.84 ± 10.18	1.05 ± 0.03	0.03 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.68	6,700,777
Gemma3-4B	skills	0.23 ± 0.07	10.38 ± 3.68	1.19 ± 0.07	0.14 ± 0.04	0.02 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.03	629,528
	mixed	0.10 ± 0.05	4.50 ± 1.64	1.06 ± 0.03	0.06 ± 0.03	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.11	2,747,439
	primitives	0.00 ± 0.00	0.41 ± 0.27	1.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.08	1,872,859
Gemma3-12B	skills	0.89 ± 0.12	63.27 ± 10.98	1.77 ± 0.13	0.47 ± 0.06	0.09 ± 0.04	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.06 ± 0.03	0.03	781,424
	mixed	0.13 ± 0.06	7.53 ± 2.67	1.11 ± 0.05	0.08 ± 0.03	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.02	461,737
	primitives	0.00 ± 0.00	0.58 ± 0.28	1.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.01	138,330
Gemma3-27B	skills	1.26 ± 0.14	140.17 ± 21.58	2.09 ± 0.15	0.62 ± 0.06	0.11 ± 0.04	0.02 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.19 ± 0.05	0.13	1,620,361
	mixed	1.29 ± 0.15	131.70 ± 18.91	2.06 ± 0.16	0.59 ± 0.06	0.12 ± 0.04	0.02 ± 0.02	0.02 ± 0.02	0.02 ± 0.02	0.14 ± 0.04	0.33	4,147,507
	primitives	0.44 ± 0.10	55.09 ± 8.64	1.23 ± 0.07	0.19 ± 0.05	0.02 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.34	4,253,730
Gemma-4-E2B	skills	1.05 ± 0.11	54.17 ± 6.39	1.86 ± 0.11	0.62 ± 0.06	0.05 ± 0.03	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.09 ± 0.04	0.16	4,758,126
	mixed	1.37 ± 0.11	104.25 ± 8.97	2.36 ± 0.14	0.75 ± 0.05	0.16 ± 0.05	0.00 ± 0.00	0.00 ± 0.00	0.02 ± 0.02	0.16 ± 0.05	0.06	1,893,335
	primitives	0.08 ± 0.04	10.08 ± 3.10	1.05 ± 0.03	0.05 ± 0.03	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.05	1,475,575
Gemma-4-E4B	skills	1.24 ± 0.12	100.58 ± 12.88	2.06 ± 0.14	0.64 ± 0.06	0.16 ± 0.05	0.00 ± 0.00	0.00 ± 0.00	0.02 ± 0.02	0.09 ± 0.04	0.25	3,753,052
	mixed	1.78 ± 0.15	216.17 ± 22.76	2.72 ± 0.21	0.69 ± 0.06	0.28 ± 0.06	0.06 ± 0.03	0.00 ± 0.00	0.02 ± 0.02	0.16 ± 0.05	0.31	4,746,369
	primitives	0.62 ± 0.11	75.73 ± 11.06	1.23 ± 0.07	0.19 ± 0.05	0.02 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.50	7,431,707
Gemma-4-31B	skills	3.21 ± 0.30	568.67 ± 70.24	4.06 ± 0.29	0.91 ± 0.04	0.50 ± 0.06	0.28 ± 0.06	0.11 ± 0.04	0.09 ± 0.04	0.30 ± 0.06	0.29	1,792,420
	mixed	3.16 ± 0.24	582.70 ± 66.09	3.89 ± 0.27	0.89 ± 0.04	0.50 ± 0.06	0.25 ± 0.05	0.06 ± 0.03	0.06 ± 0.03	0.34 ± 0.06	1.08	6,996,392
	primitives	1.45 ± 0.12	171.11 ± 16.01	1.05 ± 0.03	0.05 ± 0.03	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	2.28	12,738,072
Qwen-3.5-4B	skills	1.80 ± 0.13	324.48 ± 33.11	2.61 ± 0.17	0.78 ± 0.05	0.20 ± 0.05	0.05 ± 0.03	0.00 ± 0.00	0.02 ± 0.02	0.27 ± 0.06	0.07	1,571,941
	mixed	1.77 ± 0.10	237.41 ± 24.97	2.56 ± 0.15	0.83 ± 0.05	0.22 ± 0.05	0.02 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.28 ± 0.06	0.08	2,074,919
	primitives	0.51 ± 0.11	59.91 ± 11.67	1.09 ± 0.04	0.08 ± 0.03	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.27	5,280,187
Qwen-3.5-9B	skills	2.37 ± 0.14	373.59 ± 29.49	3.34 ± 0.21	0.89 ± 0.04	0.36 ± 0.06	0.11 ± 0.04	0.03 ± 0.02	0.02 ± 0.02	0.33 ± 0.06	0.14	1,377,070
	mixed	2.25 ± 0.17	373.58 ± 33.40	3.33 ± 0.20	0.89 ± 0.04	0.42 ± 0.06	0.11 ± 0.04	0.02 ± 0.02	0.03 ± 0.02	0.28 ± 0.06	0.42	4,076,366
	primitives	1.12 ± 0.13	121.66 ± 13.13	1.41 ± 0.12	0.23 ± 0.05	0.05 ± 0.03	0.02 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.03 ± 0.02	1.09	9,946,628
Qwen-3.5-27B	skills	3.41 ± 0.43	529.08 ± 56.37	4.28 ± 0.29	0.94 ± 0.03	0.53 ± 0.06	0.30 ± 0.06	0.06 ± 0.03	0.06 ± 0.03	0.38 ± 0.06	0.46	1,163,767
	mixed	3.65 ± 0.40	523.05 ± 46.49	4.52 ± 0.29	0.98 ± 0.02	0.61 ± 0.06	0.30 ± 0.06	0.11 ± 0.04	0.05 ± 0.03	0.38 ± 0.06	0.68	2,129,585
	primitives	1.31 ± 0.12	177.41 ± 20.04	1.59 ± 0.12	0.36 ± 0.06	0.06 ± 0.03	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.05 ± 0.03	4.09	9,302,815

Table 5: Average zero-shot NetHack performance and resource use across 14 models with both primitive and skill runs for every metric. Gray ± terms are standard errors across paired model-level means. Ratios and percent changes compare the skills average against the primitives average; lower cost and token values indicate reduced resource use. Ratios with a zero primitive baseline are undefined and shown as –.

Metric	Primitives avg	Skills avg	Skills / primitives	% change
Progression (%)	0.690 ± 0.164	1.978 ± 0.325	2.866×	+187%
Score	83.948 ± 20.157	318.268 ± 89.226	3.791×	+279%
Max DLvl	1.188 ± 0.052	2.880 ± 0.340	2.425×	+143%
L2 reach	0.133 ± 0.032	0.719 ± 0.062	5.411×	+441%
L4 reach	0.013 ± 0.005	0.286 ± 0.068	21.320×	+2032%
L6 reach	0.001 ± 0.001	0.117 ± 0.043	104.540×	+10354%
L8 reach	0.000 ± 0.000	0.036 ± 0.019	–	–
L10 reach	0.000 ± 0.000	0.004 ± 0.003	–	–
Oracle reach	0.000 ± 0.000	0.022 ± 0.008	–	–
Gnomish Mines reach	0.009 ± 0.004	0.224 ± 0.040	25.125×	+2413%
Cost / episode ($)	4.348 ± 3.556	0.590 ± 0.451	0.136×	-86%
Total tokens / episode	6,720,289 ± 1,133,932	1,741,509 ± 310,306	0.259×	-74%

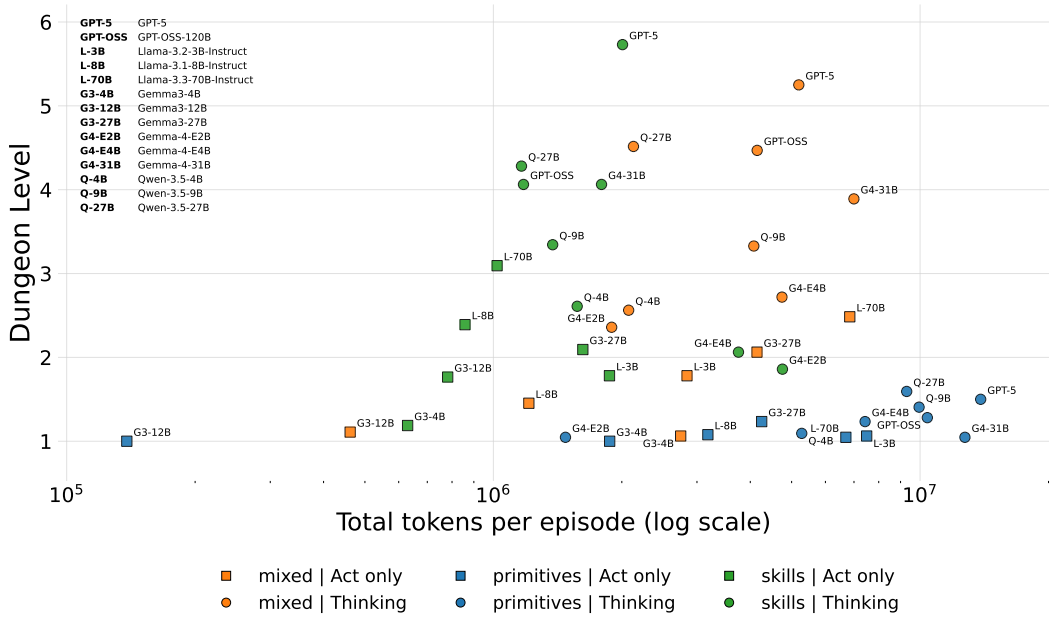

Figure 12: **Additional zero-shot NetHack tradeoff plot: average maximum dungeon level reached versus tokens per episode.** As in the main-text cost-based view, skill-based control defines a stronger performance-efficiency frontier than primitive-only control, while mixed control is typically intermediate.

Additional zero-shot tradeoff plots. For completeness, we also report the remaining pairwise zero-shot tradeoff plots relating NetHack performance to inference cost and token usage. The main text already includes maximum dungeon level versus cost (Fig. 4); here we additionally show maximum dungeon level versus tokens, progression versus cost and tokens, and score versus cost and tokens.

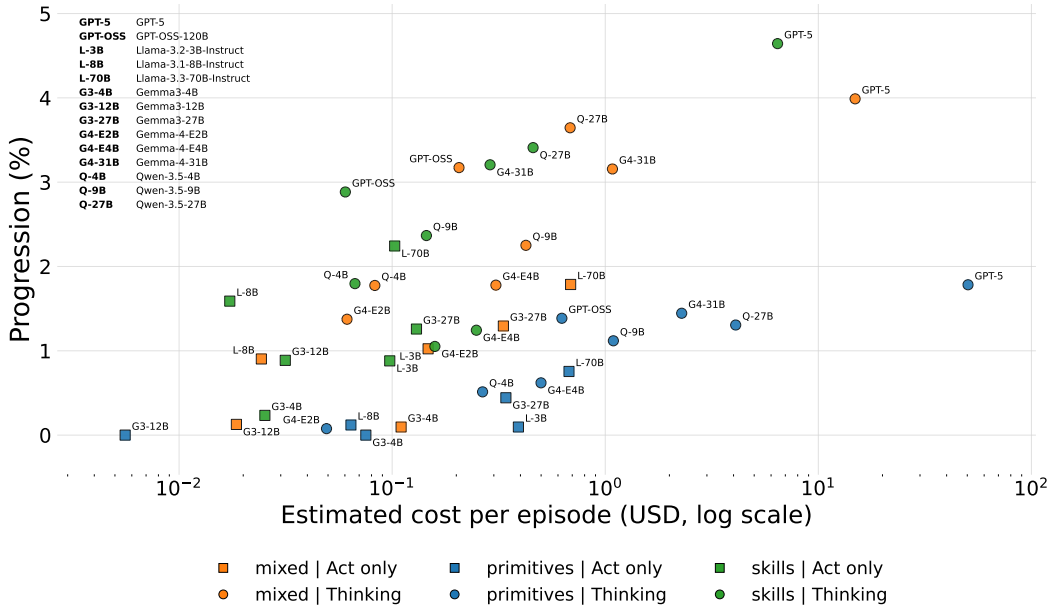

Figure 13: **Additional zero-shot NetHack tradeoff plot: progression versus inference cost per episode.** Skill-based control generally achieves higher progression at lower or comparable cost than primitive-only control.

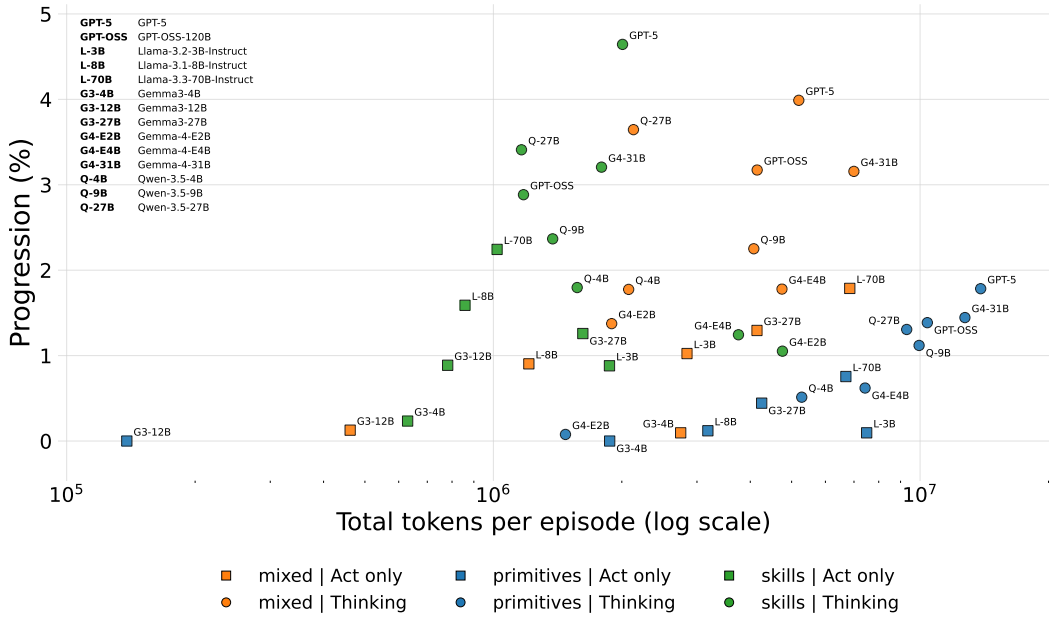

Figure 14: **Additional zero-shot NetHack tradeoff plot: progression versus token usage per episode.** Skills remain favorable when efficiency is measured in tokens rather than monetary cost.

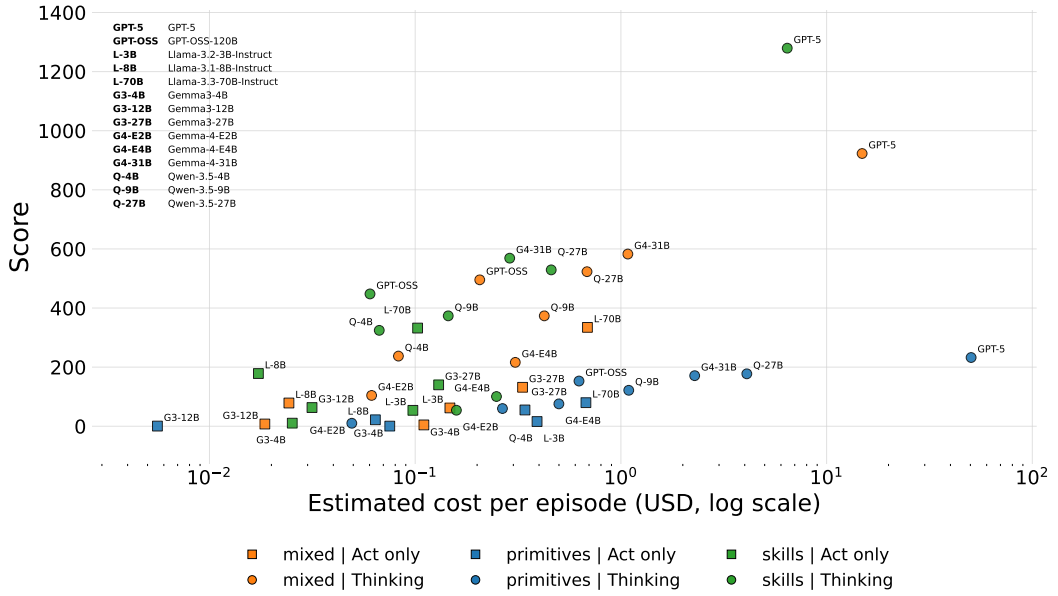

Figure 15: **Additional zero-shot NetHack tradeoff plot: score versus inference cost per episode.** Across models, skill-based control tends to produce higher scores at lower or comparable cost than primitive-only control.

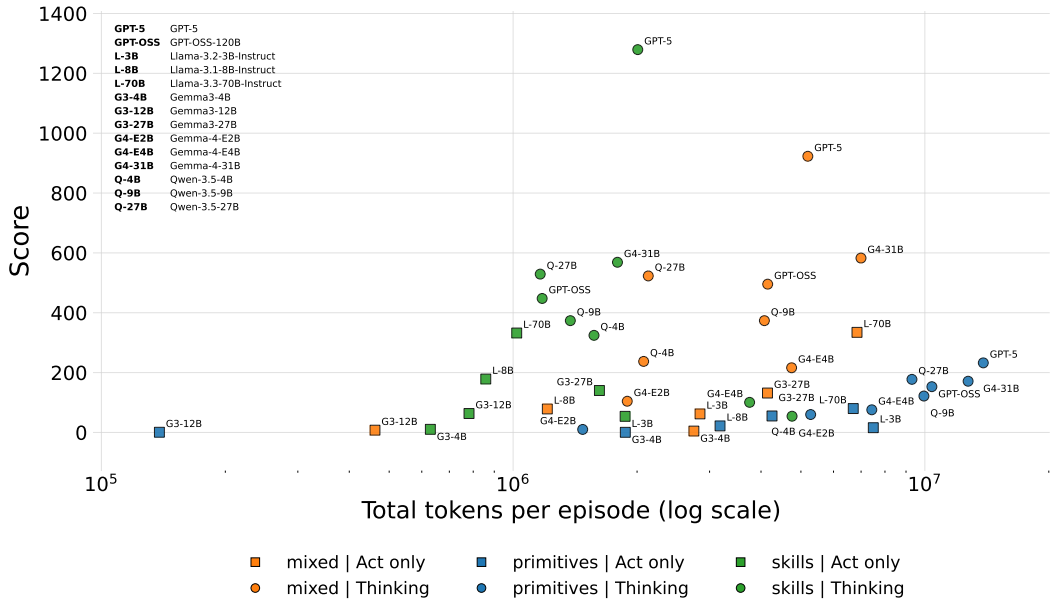

Figure 16: **Additional zero-shot NetHack tradeoff plot: score versus token usage per episode.** The same qualitative advantage of skills persists when efficiency is measured in total input and output tokens.

C.1 SENSITIVITY TO INCOMPLETE SKILL COVERAGE

We ask how skill-only and mixed controllers respond when the supplied library lacks a family of useful behaviors. Here, *coverage* refers to whether the available skills provide the behaviors needed in the situations the agent encounters. Starting from the full 78-skill library, we remove one of the five behavioral families listed in Appendix A.5 at a time, keeping the language model and runtime fixed. We evaluate zero-shot `Gemma-4-31B` on NetHack under skill-only and mixed control over 64 episodes each. In mixed control, primitive commands remain available after each removal, allowing the agent to attempt behaviors no longer supplied as skills.

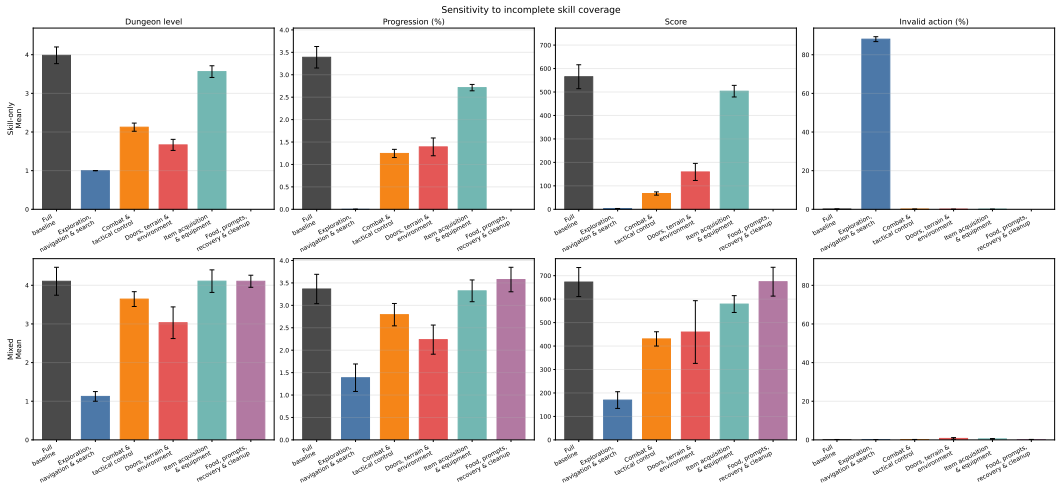

Figure 17: **Sensitivity to incomplete skill coverage for zero-shot `Gemma-4-31B` on NetHack.** The full baseline uses all 78 skills; each other condition removes the named skill family. Mixed control retains primitive actions. Bars show means over 64 episodes per condition. The comparison measures sensitivity to missing skill families and the extent to which primitive alternatives preserve performance.

The skill-only results in Fig. 17 show the importance of covering essential behaviors. Removing exploration, navigation, and search also removes staircase skills and severely restricts the agent's ability to explore and advance through the dungeon. The resulting drop in progression demonstrates dependence on coverage of these behaviors.

Mixed control lets us examine the value of supplying a skill family when primitive alternatives remain available. Family removals are generally less damaging in this setting, consistent with primitive fallback helping the controller cope with incomplete coverage. Removing exploration/navigation/search still produces the largest decrease in progression and dungeon level, followed by doors/terrain/environment and then combat/tactical control. Access to primitives therefore mitigates the loss of supplied skills, but does not guarantee successful recovery.

C.2 QUALITATIVE ANALYSIS OF MIXED CONTROL

To examine how agents move between abstraction levels, we analyze zero-shot `GPT-5` trajectories on NetHack under mixed control. We identify sequences in which primitives complete menus, recover from failed skills, or supply behaviors missing from the library, after which the agent returns to skills. Two examples illustrate this behavior (Table 6).

Table 6: **Primitive fallback in two zero-shot `GPT-5` trajectories.** Numbers are controller-decision indices in the trajectories.

Example	Skill limitation	Primitive fallback	Return to skills
Ranged combat	732: `fight_ranged` reports no ranged weapon or ammunition and executes zero environment steps.	733–735: `throw`, `i`, `east` select a javelin and kill the gas spore. 736–739: `east` moves to the dropped weapons.	740: `pickup_weapon` recovers the weapons.
Vault guard	480: `explore` executes zero environment steps. 483: the guard repeats its demand to drop the gold; no drop-gold skill is available.	484–485: `drop`, `$` drop all 650 gold pieces. 486–487: `east` follows the corridor.	488: `explore` resumes navigation.

The ranged-combat example shows recovery after an existing skill fails to complete a local subproblem. The vault example shows how primitives supply a behavior absent from the skill library. In both cases, the agent resolves the immediate problem and returns to higher-level control. These sequences support the motivation for mixed control: primitive fallback can enable the agent to move down the abstraction ladder when a skill is insufficient and back up once the problem is resolved.

Mixed control also introduces a routing problem: the controller must decide both what to do and whether to use a skill or a primitive. The vault sequence illustrates that this choice can be inefficient. Before dropping the gold, `GPT-5` opens and cancels unrelated menus, calls `explore` without advancing the environment, and moves north and back south. This is consistent with our observation that `GPT-5` sometimes switches too late or uses primitives aimlessly. The selected zero-shot examples show that useful recoveries are possible; their frequency and how RL changes these decisions require further analysis.

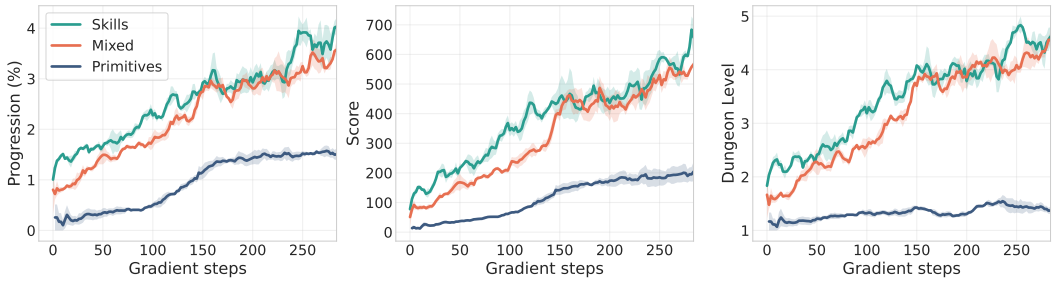

Figure 18: **Full training curves for `Llama-3.1-8B-Instruct` (gradient steps).** We plot progression (left), score (middle), and dungeon level (right) vs. gradient steps. All curves are averages across three seeds, and error bands indicate one standard error.

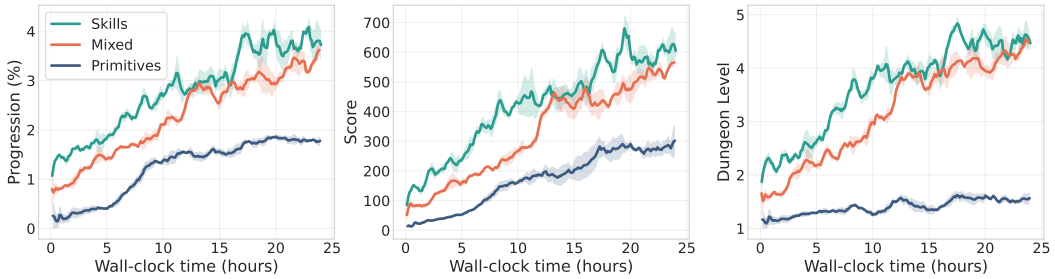

Figure 19: **Full training curves for `Llama-3.1-8B-Instruct` (wall-clock time).** We plot progression (left), score (middle), and dungeon level (right) vs. wall-clock time. All curves are averages across three seeds, and error bands indicate one standard error.

D FULL RL RESULTS

Figs. 18 and 19 and Figs. 20 and 21 give the full RL training curves for `Llama-3.1-8B-Instruct` and `Qwen-3.5-4B`, respectively, all under the three action interfaces. The first figure plots learning against gradient steps, while the second replots the same runs against wall-clock time. Together they show that the qualitative ranking is stable under both views: skill-based and mixed controllers improve faster than primitive control throughout training, and the mixed and skill interfaces maintain a large performance advantage by the end of training. The action-distribution plots in Figs. 22 to 27 complement that performance view by showing how RL changes the distribution of skills, primitives, and invalid outputs.

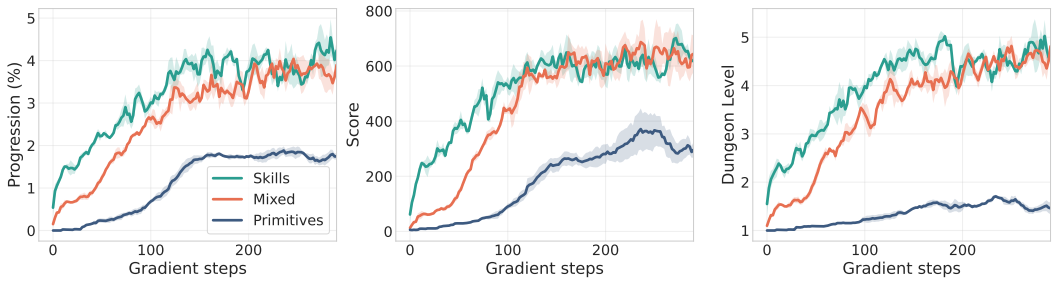

Figure 20: **Full training curves for `Qwen-3.5-4B` (gradient steps).** We plot progression (left), score (middle), and dungeon level (right) vs. gradient steps. All curves are averages across three seeds (except for mixed, which uses 2 seeds), and error bands indicate one standard error.

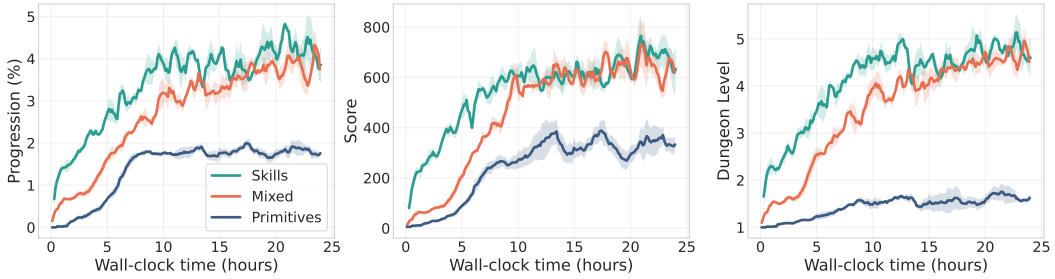

Figure 21: **Full training curves for `Qwen-3.5-4B` (wall-clock time).** We plot progression (left), score (middle), and dungeon level (right) vs. wall-clock time. All curves are averages across three seeds (except for mixed, which uses 2 seeds), and error bands indicate one standard error.

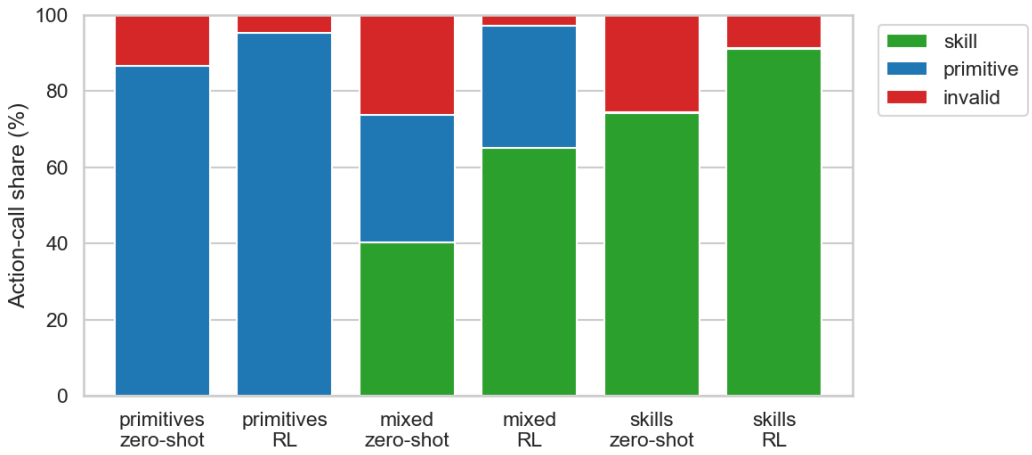

Figure 22: **Aggregate action-type mix before and after RL on Llama-3.1-8B-Instruct.** Bars show the share of parsed calls attributed to skills, primitives, and invalid outputs, aggregated across all evaluation episodes in each stage. RL shifts the mixed controller toward more successful skill use and sharply reduces invalid calls, but does not eliminate primitive fallback: the mixed primitive share remains close to one third of all calls.

D.1 Action-distribution analysis for Llama-3.1-8B-Instruct

To better understand how RL changes the use of the abstraction ladder, we compare parsed action distributions before and after RL for Llama-3.1-8B-Instruct. Figs. 22 and 23 summarize the aggregate and per-episode shift between skills, primitives, and invalid outputs, while Figs. 24 to 27 break that shift down into the most frequent calls in the mixed, skill-only, and primitive-only interfaces. Three patterns are especially clear. First, mixed-control RL does not simply eliminate primitive use: by aggregate share, primitive calls remain almost unchanged in the mixed interface (33.3% in zero-shot vs. 32.0% after RL). Second, the typical mixed episode still becomes much more skill-heavy after training. The median skill fraction rises from 41.1% to 77.7%, while the median primitive fraction falls from 32.9% to 20.2%. Third, RL concentrates behavior around a small set of simple but important routines, while sharply reducing malformed or otherwise invalid calls.

D.2 OOD MiniHack transfer for Llama-3.1-8B-Instruct

Fig. 28 compares MiniHack success before and after RL training on NetHack. RL improves average success for both action interfaces, but the gains differ across tasks. For primitive control, improvements are concentrated on the Corridor tasks, with one additional success on `Quest-Medium` after RL. For skill-based control, RL brings Corridor success close to saturation and improves `Quest-Easy` and `Quest-Medium`. However, `Quest-Hard` remains unsolved, and `WoD-Hard` success decreases. Averaged across tasks, success rises from 56.5% to 61.8% for skills and from 12.2% to 24.8% for primitives. Thus, primitives show a larger average absolute improvement, while skills retain substantially higher success overall.

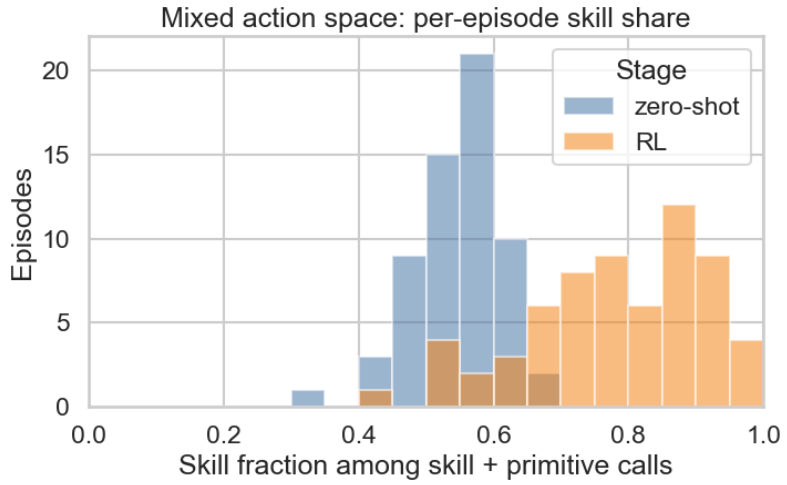

Figure 23: **Per-episode skill and primitive fractions before and after RL.** Each histogram is computed over per-episode call fractions derived from parsed action strings. We do not show primitive-only and skill-only interfaces, because the distributions collapse to 0 or 1 by construction. The important change is in the mixed interface: RL leaves the aggregate primitive share nearly unchanged (33.3% to 32.0%), but the typical episode becomes much more skill-heavy because the model outputs fewer invalid actions. The median mixed-episode skill fraction increases from 41.1% to 77.7%, while the median primitive fraction falls from 32.9% to 20.2%.

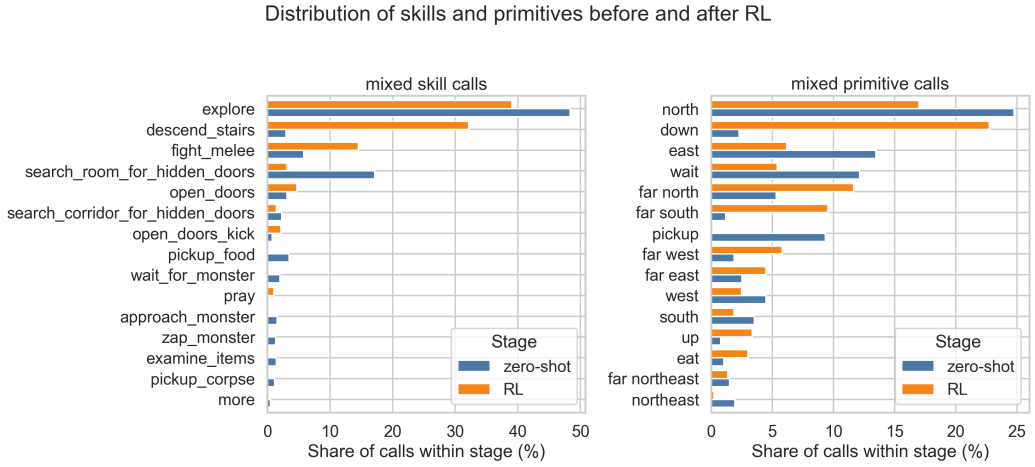

Figure 24: **Top mixed-controller skills and primitives before and after RL.** The left panel shows skill calls and the right panel shows primitive calls within the mixed interface. In each panel, we plot the top-k call types with $k = 15$, selected and ordered by the sum of their before-RL and after-RL call counts. After RL, mixed control concentrates its skill usage on a small core of reusable routines, especially `explore`, `descend_stairs`, `fight_melee`, `open_doors`, and hidden-door search, while still retaining a nontrivial set of movement and staircase primitives.

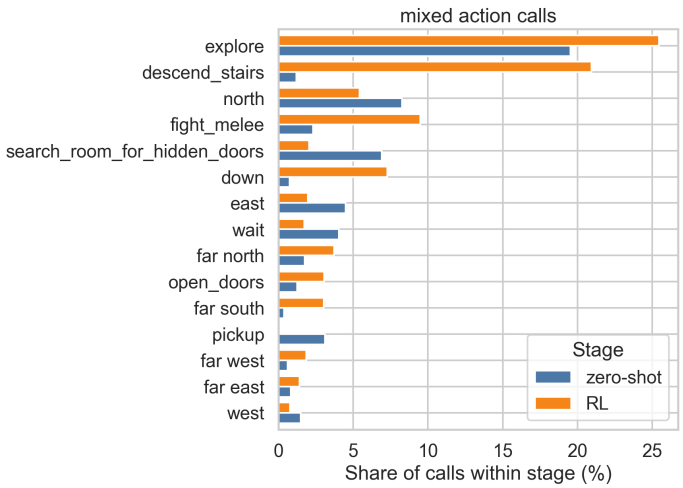

Figure 25: **Top action calls in the mixed interface before and after RL.** We plot the top-k actions with $k = 15$, selected and ordered by the sum of their before-RL and after-RL call counts within the mixed interface. When skills and primitives are pooled together, `explore` remains the single most common action both before and after training. RL elevates high-value skills such as `descend_stairs` and `fight_melee`, while several directional primitives decline in relative share, consistent with the controller relying more often on reusable skills rather than long sequences of local movement decisions.

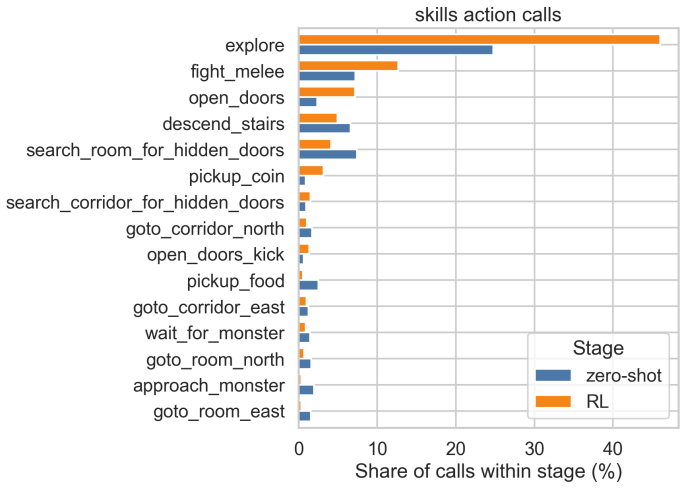

Figure 26: **Top action calls in the skill-only interface before and after RL.** We plot the top-k actions with $k = 15$, selected and ordered by the sum of their before-RL and after-RL call counts within the skill-only interface. Skill-only RL becomes even more concentrated on a handful of reusable routines, with `explore` dominating and combat, door handling, staircase navigation, and hidden-door search accounting for much of the remaining mass. This suggests that much of the learned improvement comes from invoking a small core of robust behaviors more effectively, rather than from broad uniform use of the entire library.

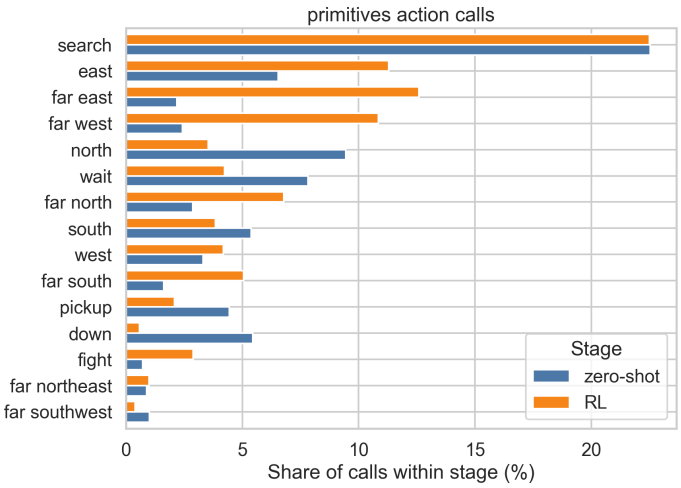

Figure 27: **Top action calls in the primitive-only interface before and after RL.** We plot the top-k actions with $k = 15$, selected and ordered by the sum of their before-RL and after-RL call counts within the primitive-only interface. Primitive-only control remains dominated by local movement and `search`. Training changes the mix of directions and reduces invalid calls (Fig. 22), but it does not create the same kind of reusable behavioral compression available in the skill-based interfaces.

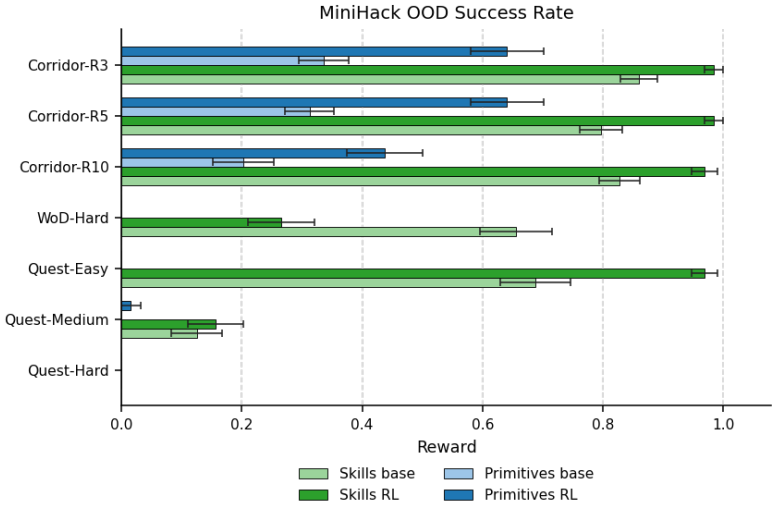

Figure 28: **OOD MiniHack transfer before and after NetHack RL.** Bars compare the Llama-3.1-8B-Instruct for primitives and skills before RL and after RL on held-out MiniHack tasks. We report average success rates. Both interfaces transfer from NetHack to MiniHack, but RL changes them differently: primitive gains are concentrated on the Corridor tasks, while skill-based RL pushes the Corridor family to near-perfect success and also improves the Quest tasks. The main exception is `WoD-Hard`, where the skill controller becomes worse after RL.