GenHRL: Generative Hierarchical Reinforcement Learning

Anonymous Author(s)
Affiliation
Address
`email`

Abstract: Defining effective multi-level skill hierarchies and their corresponding learning objectives is a core challenge in robotics and reinforcement learning. Large Language Models (LLMs) offer powerful new capabilities for tackling this challenge through automated generation and reasoning. This paper introduces GenHRL, an LLM-driven framework that automates the pipeline from high-level natural language task descriptions to learned hierarchical skills. GenHRL autonomously generates: (1) task-specific simulation environments, (2) multi-level skill decompositions, and (3) executable code defining intrinsic reward and termination functions for each skill. This automation avoids the need for manual reward engineering, predefined skill sets, offline datasets, and enables end-to-end hierarchical policy learning via standard reinforcement learning algorithms. Empirical evaluations on complex robotic humanoid simulation tasks demonstrate that GenHRL significantly enhances learning efficiency and final performance compared to non-hierarchical baselines.

Keywords: Reinforcement Learning, Hierarchical Reinforcement Learning, Generative AI, Robotics.

1 Introduction

A fundamental goal in Artificial Intelligence (AI) and robotics is to develop agents capable of acting competently in complex, unstructured environments, adapting to new situations and tasks [1]. Humans exhibit this level of task flexibility, believed partly to stem from organising behaviour into hierarchies of temporally extended actions or skills [2, 3]. This hierarchical structure allows us to decompose complex problems into manageable sub-tasks, reuse learned skills across different contexts (e.g., grasping a cup vs. grasping a tool), and learn new tasks more quickly by composing existing competencies [4].

Hierarchical Reinforcement Learning (HRL) provides a framework for structuring agent behaviours, mirroring the hierarchical organisation believed to underpin flexible intelligence [1]. By decomposing tasks and learning reusable skills, HRL offers potential benefits in sample efficiency, exploration, and generalisation, particularly for long-horizon robotic control problems [5, 6, 7]. Indeed, recent work focusing specifically on multi-level skill hierarchies has demonstrated that such structures, when learned effectively, can substantially accelerate adaptation and generalisation to new tasks [7], underscoring the potential value of deep hierarchical representations for achieving robust robot autonomy.

However, realising the full potential of deep HRL is currently hindered by the significant challenge of automating hierarchy creation and definition. Manually designing multi-level skill structures with appropriate intrinsic objectives is exceptionally difficult and requires extensive expertise [8, 9]. While automated skill discovery methods can learn reusable behaviours from data, they often require substantial environment experience [9, 10, 7].

Large Language Models (LLMs) present a powerful tool for potentially overcoming this automation bottleneck. Their ability to interpret natural language and generate structured outputs, including code, suggests they could automate parts of the design process [11, 12, 13]. While recent works have shown promise in using LLMs to automate specific parts of the RL workflow, such as reward design for predefined tasks or environment generation [14, 15, 16, 17], the crucial challenge of end-to-end automation directly from a high-level language instruction remains. Specifically, the autonomous interpretation of a language goal to derive not only the environment configuration but also the appropriate multi-level skill hierarchy and the complete set of rewards and success objectives required to learn each of the skills within that hierarchy, has not been fully addressed.

Here we introduce GenHRL, a system designed to leverage LLMs to orchestrate a fully automated pipeline that translates a single, high-level natural language task description into a learned, multi-level skill hierarchy ready for execution and, potentially, transfer. It does not require pre-defined skills, large demonstration datasets, or manual specification of rewards or hierarchical structure. The core capability of GenHRL lies in the LLM's automated generation of: (1) a task-relevant simulation environment, (2) a multi-level decomposition of the task into skills, and (3) the executable code defining the reward signals and completion conditions for each skill in the hierarchy. These generated components directly enable subsequent automated skill learning using standard reinforcement learning algorithms.

We demonstrate GenHRL on a simulated humanoid robot, showing its ability to generate and learn hierarchical skills for complex obstacle interaction tasks. We validate that the automatically generated hierarchy leads to significantly improved learning efficiency and final performance compared to standard reinforcement learning methods trained with the same objectives. We also demonstrate how the learned skill hierarchy can be flexibly composed to zero-shot complete the original task. Finally, we describe how end-to-end skill generation enables LLM-driven training augmentation, which could lead to a hierarchical library of re-useable skills. This work represents a step towards scalable, language-driven acquisition of complex, hierarchical behaviours in robots.

2 Background

This section briefly introduces the core concepts essential for understanding our method: the relevant aspects of the Isaac Lab simulation environment, the fundamentals of Reinforcement Learning (RL), and Hierarchical RL (HRL) using the options framework.

Reinforcement learning provides a framework for an agent to learn optimal behaviour through interaction with an environment [18, 19]. This interaction is typically formalised as a Markov Decision Process (MDP), defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$. Here, $\mathcal{S}$ represents the set of possible states, $\mathcal{A}$ is the set of primitive actions available to the agent, $\mathcal{P}(s'|s,a)$ defines the probability of transitioning to state $s' \in \mathcal{S}$ from state $s \in \mathcal{S}$ after taking action $a \in \mathcal{A}$, $\mathcal{R}(s,a,s')$ gives the immediate expected reward for this transition, and $\gamma \in [0,1]$ is the discount factor. The agent learns a policy $\pi : \mathcal{S} \to P(\mathcal{A})$, mapping states to a probability distribution over actions, with the goal of maximizing the expected cumulative discounted return, $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$.

In the context of this work, where skill-specific objectives are automatically generated (Section 3.2), it is useful to distinguish the underlying physical system from the specific objective being pursued. We therefore refer to the core environment as being defined by the state space $\mathcal{S}$, the primitive action space $\mathcal{A}$, the transition dynamics $\mathcal{P}$, and an initial state distribution μ. A specific task or skill objective is then defined by pairing this environment with a reward function ($\mathcal{R}$ or skill-specific $\mathcal{R}_z$) and a discount factor γ.

Hierarchical Reinforcement Learning (HRL) extends reinforcement learning by introducing temporal abstraction, enabling agents to operate using higher-level actions or skills built upon lower-level ones [1]. The options framework [20] is a common way to formalize these temporally extended actions, which we refer to as skills. A skill z is defined by a tuple $(\mathcal{I}_z, \pi_z, \beta_z)$. Where, $\mathcal{I}_z \subseteq \mathcal{S}$ is the initiation set, defining states where skill z can begin; π_z is the policy followed while skill z is executing; $\beta_z : \mathcal{S} \to [0, 1]$ defines the probability of skill z terminating in a given state.

In a multi-level hierarchy, as employed in our work, skills can operate at different layers. A higher-level skill z_i might have an intra-option policy π_{z_i} that selects among a set of lower-level skills $\{z_{i,j}\}$, while only the lowest-level skills have policies $\pi_{z_{i,j,\dots}}$ selecting primitive actions $a \in \mathcal{A}$. Learning the policy π_z for each skill z typically requires its own objective, often defined via a skill specific intrinsic reward function $\mathcal{R}_z$. The challenge lies in discovering or defining the entire hierarchy – the structure, the skills z at each level, and their corresponding components $(\mathcal{I}_z, \pi_z, \beta_z, \mathcal{R}_z)$ – necessary to solve a complex task effectively [9, 7]. Automating this definition process is the focus of our work.

2.1 Isaac Lab Simulation Environment

Isaac Lab is a high-performance robotics simulation platform designed for efficient RL training [21]. Its architecture is particularly relevant to our work due to its modularity and Python API. Environments and tasks within Isaac Lab can be defined programmatically. This includes the physical scene configuration (robot models, objects, layout) which is specified entirely via code.

Crucially, the reinforcement learning task logic is also managed through distinct, code-configurable components. For instance, the calculation of rewards is handled by a Reward Manager ('RewardManager'), termination conditions are evaluated by a Termination Manager ('TerminationManager'), and environment resets or specific state initializations (often triggered by termination events) can be controlled via an Event Manager or similar reset logic configurations. Critically, this manager-based system can be fully configured through Python code. This provides the essential interface for our approach. It allows the executable code snippets generated by our LLM to compose an MDP such that each skill can be trained using reinforcement learning.

3 Method

Our method, illustrated in Figure 1, translates a high-level natural language task description, g, into a learned multi-level hierarchy of executable skills. An LLM orchestrates the process by: (1) decomposing the task and configuring the simulation environment, (2) generating formal specifications (rewards, terminations, initiations) for skills at each level of the hierarchy, and (3) automatically running a hierarchical reinforcement learning phase which trains each skill policy π_z at all levels.

3.1 Automated Task Design and Required Skill Generator

The process begins with a natural language task description g. An LLM[1] receives this description and, conditioned on general Isaac Lab simulation constraints, performs two key initial generation steps. First, it automatically decomposes g into a hierarchy[2] of constituent skills. This decomposition results in a structured set, $\mathcal{D}_Z$, containing natural language descriptions for skills across the different levels of abstraction. Concurrently, based on the original goal g, the LLM also generates the Isaac Lab configuration code required to instantiate a task-specific simulation environment, denoted $Env(g)$, see Figure 2 for two examples. This generated code defines the scene and necessary components, enabling subsequent skill training within an environment precisely tailored to the specified task. The outputs of this stage are thus the structured skill descriptions $\mathcal{D}_Z$ and the configured environment $Env(g)$.

3.2 Automated Per-Skill Function Design

Following the hierarchical decomposition, each natural language skill description $d_z \in \mathcal{D}_Z$ must be translated into formal specifications usable for RL training. To achieve this, the LLM is prompted

[1]Gemini 2.0 flash thinking used for all experiments.

[2]We create a three level hierarchy in this work, however higher levels are possible.

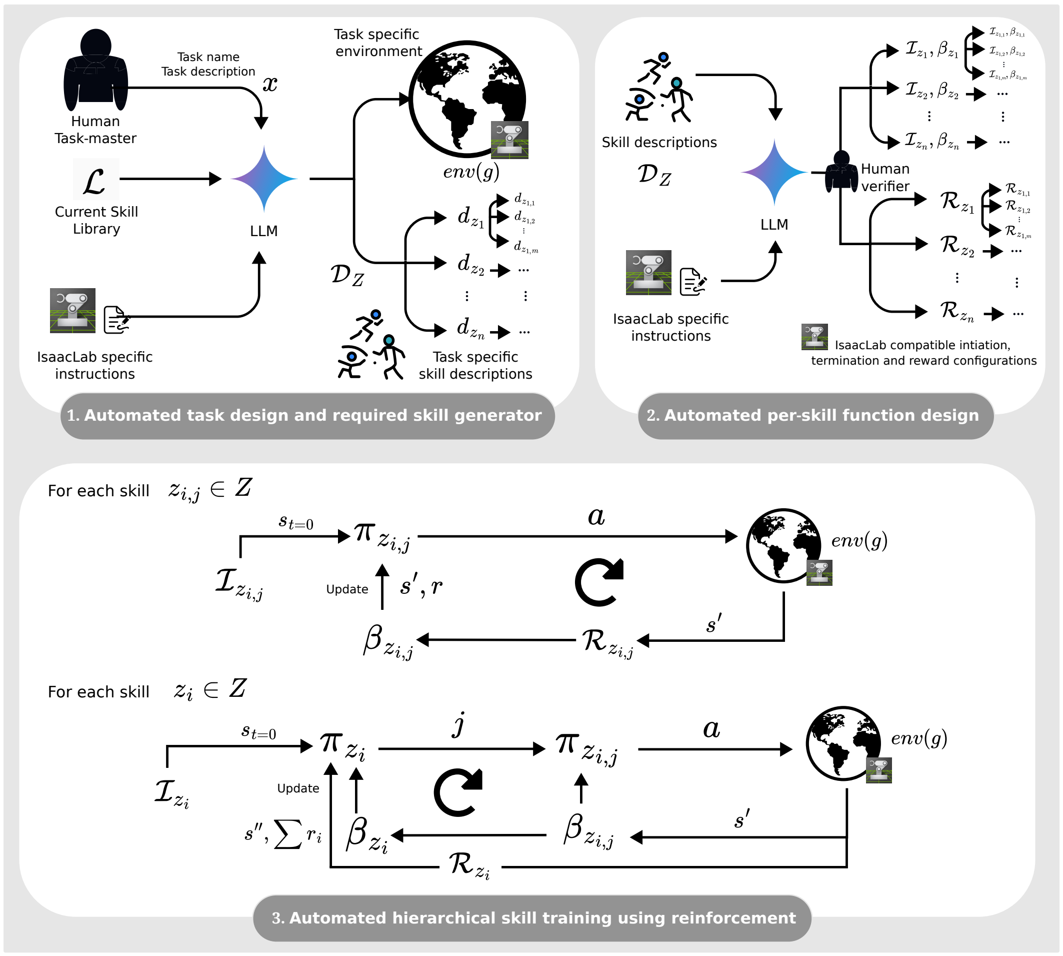
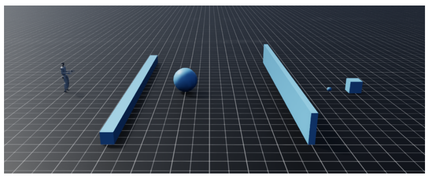

Human Task-master
Task name
Task description
x
Task specific environment
$env(g)$
$\mathcal{L}$
Current Skill Library
LLM
IsaacLab specific instructions
$\mathcal{D}_Z$
Task specific skill descriptions
1. Automated task design and required skill generator
Skill descriptions
$\mathcal{D}_Z$
Human verifier
IsaacLab compatible intiation, termination and reward configurations
2. Automated per-skill function design
For each skill $z_{i,j} \in Z$
For each skill $z_i \in Z$
Update
3. Automated hierarchical skill training using reinforcement

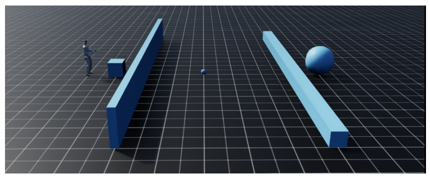
Figure 1: Overview of our method for automated multi-level hierarchical skill generation and learning. (1) Given a language goal g, an optional skill library $\mathcal{L}_z$, and simulator constraints, an LLM generates a task-specific environment $env(g)$ and a set of hierarchical skill descriptions $\mathcal{D}_Z$. (2) For each skill description (e.g., $d_{z_i}, d_{z_{i,j}} \in \mathcal{D}_Z$), the LLM (optionally mediated by a human verifier) generates Isaac Lab compatible code defining initiation conditions ($\mathcal{I}.$), termination conditions ($\beta.$), and intrinsic reward functions ($\mathcal{R}.$) for each skill within the hierarchy. (3) Automated hierarchical skill training occurs: high-level policies π_{z_i} learn to select (j) lower-level skills $z_{i,j}$, whose policies $\pi_{z_{i,j}}$ learn to select primitive actions using RL, guided by their respective generated rewards and terminations within the environment $env(g)$.

with relevant context, including the skill description d_z, details of the generated environment $Env(g)$, the skill's position within the hierarchy, and Isaac Lab API requirements (e.g., function signatures for reward and termination managers). Based on this comprehensive prompt, the LLM generates the necessary Python code snippets defining the skill's objectives and boundaries, as detailed below.

Intrinsic Reward Function Generation. The LLM generates code defining the intrinsic reward function $\mathcal{R}_z : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ for each skill z. This generated code is designed to be directly integrated into Isaac Lab's reward manager configuration ('RewardManager'). To facilitate dense learning signals, the LLM defines one main reward term reflecting the core objective of d_z, alongside several reward shaping terms. These are combined with predefined shaping terms using a template structure. Within this structure, each generated reward term is normalised, and assigned LLM-specified weights to control their relative importance.



Standard Obstacle Course: "The robot should jump over a low wall, push a large sphere into a high wall to knock it down and pass over it. The robot should then walk to a small sphere and kick it past a block. Finally the robot should walk to the block and jump onto it."



Alternate Obstacle Course: "The robot should jump onto a block so that it can jump over a high wall. The robot should then kick a small sphere into a low wall. The robot should then jump over the low wall and touch a large sphere."

Figure 2: Two examples of generated environments. The first is the standard obstacle course we use in this work.

Termination Condition Generation. The LLM generates code for the termination function $\beta_z : \mathcal{S} \to [0,1]$, specifically defining the success conditions based on the skill description d_z. Standard failure conditions (e.g., robot falling, timeouts) are handled separately by a common wrapper applied to all skills.

Initiation Set Handling for Training. While the initiation set $\mathcal{I}_z \subseteq \mathcal{S}$ formally defines all states where skill z might start, our primary focus during the learning phase (Section 3.3) is to ensure the initiation state from the current task roughly translates to the termination state of the previous. This means that instead generating complex state-based logic for $\mathcal{I}_z$, we leverage Isaac Lab's event-driven, manager-based system. We configure the training such that the successful termination states of the immediately previous skill z_{prev} (as determined by $\beta_{z_{prev}}$), become the initiation states in the training episode for the subsequent skill z, see the black arrows in Figure 3 for visualisation. This procedural approach enforces the intended decomposition during training. However, during full task execution, we allow all skills to be initiated from any state $s \in \mathcal{S}$.

Optional Human Verification. As illustrated in Figure 1, the skill function definitions $(\mathcal{I}_z, \mathcal{R}_z, \beta_z)$ can optionally pass through a human verification step. This allows for inspection and refinement of the LLM's code outputs, ensuring correctness or alignment before committing to computationally intensive reinforcement learning training. All modifications made during verification for our experiments are documented in Appendix A.

The outcome of this stage is a complete set of programmed definitions for each skill – comprising the intrinsic reward functions $\mathcal{R}_z$, termination conditions β_z, and initiation logic $\mathcal{I}_z$ – derived from the hierarchical language descriptions $\mathcal{D}_Z$. These generated components provide the necessary structure and learning signals for the subsequent hierarchical skill policy π_z learning phase (Section 3.3).

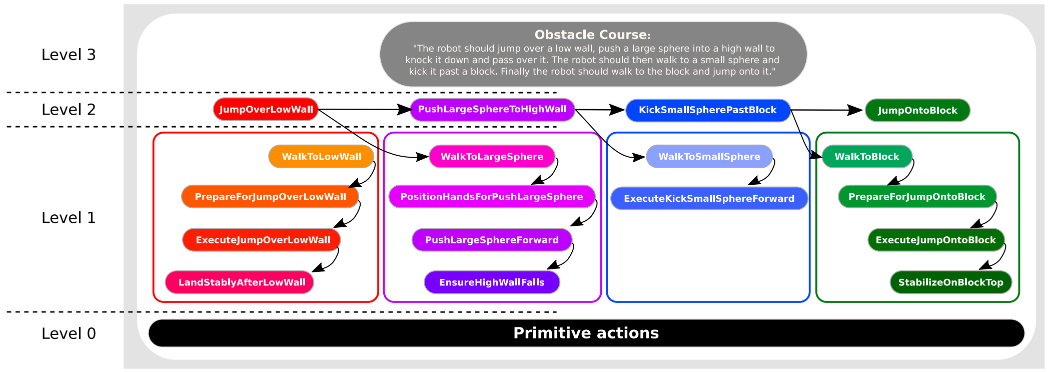

Level 3
Level 2
Level 1
Level 0
Obstacle Course:
"The robot should jump over a low wall, push a large sphere into a high wall to knock it down and pass over it. The robot should then walk to a small sphere and kick it past a block. Finally the robot should walk to the block and jump onto it."
JumpOverLowWall
PushLargeSphereToHighWall
KickSmallSpherePastBlock
JumpOntoBlock
WalkToLowWall
PrepareForJumpOverLowWall
ExecuteJumpOverLowWall
LandStablyAfterLowWall
WalkToLargeSphere
PositionHandsForPushLargeSphere
PushLargeSphereForward
EnsureHighWallFalls
WalkToSmallSphere
ExecuteKickSmallSphereForward
WalkToBlock
PrepareForJumpOntoBlock
ExecuteJumpOntoBlock
StabilizeOnBlockTop
Primitive actions

Figure 3: An example of the skill hierarchy at three levels generated by the LLM for our standard obstacle course description. The black arrows indicate the typical sequential flow during training or execution. Depth-first training would proceed by learning all level 1 skills under "JumpOverLowWall", then the parent level 2 skill "JumpOverLowWall", before moving to the next level 2 branch.

3.3 Automated Hierarchical Skill Training

This stage learns the intra-option policies π_z for all skills $z \in Z$ using the generated specifications and environment. Learning proceeds in a depth first search manner, see Appendix **??** for more details. The skills at the lowest level of the hierarchy which only select primitive actions $a \in \mathcal{A}$, are trained using standard reinforcement learning algorithms[3]. Training episodes initiate in states $s_0 \in \mathcal{I}_{z_{i,j}}$ and aim to maximise the expected return defined by the LLM-generated intrinsic reward $\mathcal{R}_{z_{i,j}}$: $\mathbb{E}[\sum \gamma^k \mathcal{R}_{z_{i,j}}(s_k, a_k, s_{k+1})]$. Training concludes when a predefined number of success states $w_{z_{i,j}}$ have been reached. For skills at higher levels, the actions available for the skill policy π_{z_i} are the sub-skill indices, $j \in \{1, \dots, m_i\}$, each representing the execution of the trained policy $\pi_{z_{i,j}}$. This sub-skill runs until its termination $\beta_{z_{i,j}}$ triggers, returning a final state s''. The policy π_{z_i} is updated based on the reward signal derived from its own intrinsic reward $\mathcal{R}_{z_i}$ not on the called $\mathcal{R}_{z_{i,j}}$. Training episodes start in $\mathcal{I}_{z_i}$ and end according to β_{z_i}.

At the task level we train a policy π_g which selects among the high level skill indices $i \in \{1, ..., n\}$ and in turn learns to optimise the task reward $\mathcal{R}_g$: $\mathbb{E}[\sum \gamma^k \mathcal{R}_g(s_k, a_k, s_{k+1})]$.

4 Experiments

We evaluate GenHRL's ability to automatically generate and learn multi-level skill hierarchies from language for complex robotic tasks. Our experiments aim to answer: (1) Does the automatically generated hierarchy increase skill acquisition efficiency over flat structures? (2) Does the full GenHRL pipeline enable solving long-horizon tasks specified via language? (3) Can the learned skills be composed zero-shot to solve the original task?

4.1 Experimental Setup

Simulation Environment. All experiments are conducted in simulation using Isaac Lab [21], leveraging its efficient parallel simulation capabilities. We use a model of the Unitree G1 humanoid robot, a high-DoF platform suitable for complex locomotion and manipulation tasks.

Task and Skill Generation. We provide GenHRL with a single high-level language description of the standard humanoid obstacle course shown in Figure 2. GenHRL then generates the task-specific environment configuration code for Isaac Lab, a three-level skill hierarchy decomposition (visualised in Figure 3), and the corresponding intrinsic reward and termination function code for each skill in the hierarchy.

Baseline Comparison. We compare GenHRL against a standard PPO baseline. To ensure a fair comparison, the baseline agent is trained in the same environment generated by GenHRL's LLM for the obstacle course task, with the equivalent reward functions and success criteria. For skill efficiency comparisons (Section 4.2), the baseline uses the specific intrinsic skill rewards. For full task comparisons (Section 4.3), the baseline uses the overall task success rewards. In both cases, the baseline learns a single policy mapping observations directly to primitive actions without access to the hierarchy.

4.2 Skill Acquisition Efficiency

To quantify the benefits of the hierarchical structure for learning efficiency on the standard obstacle course task, we first analyse the efficiency of acquisition of the level 2 skills (z_i) as shown in Figure 3. We compare GenHRL and SKRL PPO over five independently seeded experiments. Figure 4 presents the success rates as each agent learns these level 2 skills within the standard course environment.

[3]In all our experiments, we use Proximal Policy Optimisation (PPO) [22].

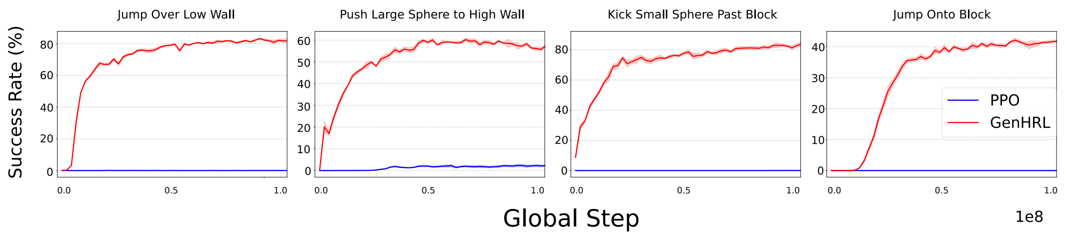


Figure 4: Success rate comparison of GenHRL framework against a baseline PPO. The solid lines show means and the shading shows the standard error, computed over five seeds.

4.3 Hierarchical Task Completion

Having demonstrated the skill acquisition efficiency improvements, we next evaluate the ability of GenHRL over the full original task g. For this experiment we train a task level policy π_g with access to the learned skills using reward functions generated from GenHRL. We track a reward which is a measure of completion of each obstacle Figure 5 demonstrates GenHRL significantly outperforms the baseline PPO.

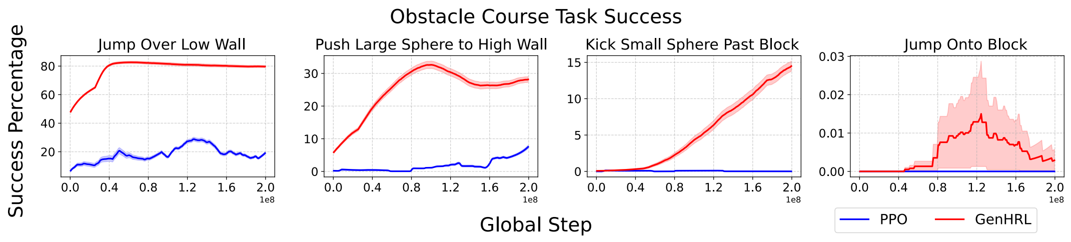


Figure 5: Success rates of each obstacle from an overall task learned policy, comparing GenHRL with a baseline PPO. Solid lines show the mean average and the shaded areas show the standard error over four seeds. Obstacle 1-4 refers to overcoming the obstacles of each skill, before the task terminates.

4.4 Zero-Shot Skill Composition

Next we evaluate the ability to zero-shot compose the level 1 skills in the original task without any retraining or fine-tuning. These skills are executed in the order decided by GenHRL's task decomposition and executed for appropriate time-steps each. We evaluate how many obstacles have been completed in each zero-shot rollout and report these results in Table 1.

Interestingly, the fixed skill sequence determined by GenHRL directly, outperforms the learned task-level policy π_g. We suspect the learned policy becomes overly risk-averse during training; large negative rewards associated with difficult or failure-prone skills may lead π_g to prefer safer, stabilising actions over attempting necessary but challenging sub-tasks. The fixed sequence bypasses this issue by mandating execution.

Zero-shot completion rates			
Low Wall	Large Sphere	Small Sphere	Block
94% (5.5)	62% (8.4)	40% (7.1)	30% (7.1)

Table 1: Success percentages of zero-shot composition of level 1 skills in the obstacle environment. Percentages are the means of five seeds. Brackets show the standard deviations. The percentages include all previous tasks, such that the 30% task success of finishing on the block, means that 30% of runs completed all obstacles.

5 Related Work

Hierarchical Skill Discovery: Early work on automated skill discovery focused on learning single-layer options through intrinsic motivation or sub-goal identification. Methods like diversity-based exploration [9], empowerment maximization [23], feudal approaches [24, 25] and leveraging state novelty or bottlenecks [26, 27]. These methods learn reusable skills without explicit task rewards but often require extensive environment interaction or access to the state transition graph. While gradient-based approaches like Option-Critic [28] enabled end-to-end learning, they often struggle with skill collapse in sparse reward settings [29]. There exist few multi-level discovery methods in the literature [6, 30, 31, 32, 7]; however, only Cannon and Şimşek's Fracture Cluster Options (FraCOs) [7] demonstrate effective generalisation improvements when the hierarchy is transferred across tasks. Furthermore, most of these discovery methods, including FraCOs, require substantial interaction with the environment prior to decomposing the task into a skill hierarchy.

LLMs for reinforcement learning Automation: LLMs have shown promise in automating various components of the reinforcement learning pipeline, leveraging their capabilities in language understanding, reasoning, and code generation. Recent works include generating reward functions for flat policies through code synthesis [15, 16], creating training environments or tasks from text description [14], and acting as high-level planners or controllers by selecting among pre-learned skills [33]. Voyage [34] also uses LLM's to define a single level of hierarchical skills however, this is also only demonstrated in a structured minecraft environment. MaestroMotif [17] uses LLMs to design skill rewards but requires per-skill human descriptions and offline observational datasets. Other related work uses LLMs to generate policy sketches or code-based policies [35]. While all these approaches automate individual aspects of hierarchical reinforcement learning, they leave critical gaps addressed by GenHRL; none decompose multiple-levels of skill abstraction or zero-shot skill execution. Most also require extensive offline datasets to learn skills.

6 Discussion

GenHRL introduces a framework for hierarchical reinforcement learning that automates the generation of multi-level skill hierarchies directly from high-level language descriptions. By leveraging LLMs to define task-specific environments, decompose tasks, and generate executable intrinsic objectives (rewards and terminations) for each skill, our system facilitates end-to-end learning without manual reward engineering or reliance on predefined skill libraries or offline datasets. This automation enables GenHRL to effectively tackle complex, long-horizon robotic tasks, such as the humanoid obstacle course, where standard non-hierarchical reinforcement learning approaches typically falter due to sparse rewards, exploration challenges and multiple stage objectives. The demonstrated improvement in learning efficiency, final task performance and zero-shot composition underscores the benefit of imposing hierarchical structure derived directly from language.

A key outcome of the GenHRL pipeline is the creation of a library of temporally abstract skills. Our experiments show that these learned skills can be composed zero-shot to solve the original task. However, we have not extended this work to demonstrate zero-shot transfer to variations of the original task. However, we believe that this ability to automatically generate multi-level hierarchies is a critical step towards forming a library of reusable skills. For instance, the generated skill library could be exploited to systematically propose and learn increasingly complex behaviours, potentially drawing parallels with programmatic skill synthesis approaches like DreamCoder [**?**], but grounded in embodied interaction. Such "dreaming" or self-improvement cycles, driven by composing existing skills to solve LLM-generated hypothetical tasks, could significantly enhance an agent's capabilities without constant environmental interaction. Positioned within the broader HRL landscape, GenHRL contributes by automating the challenging design process of multi-level hierarchies and demonstrating the tangible benefits of this structure for learning efficiency, paving the way for more scalable and adaptable reinforcement learning agents.

7 Limitations and Future Work

While GenHRL demonstrates a promising direction, several limitations warrant discussion and offer avenues for future research.

LLM Reliability and Verification: The quality and correctness of the LLM-generated skill decompositions, intrinsic rewards, and termination conditions are crucial for successful learning. In our experiments, while the LLM provided a very strong starting point, in some cases a human verification step was necessary to refine generated code, particularly for complex dynamic skills like jumping, and to ensure alignment with intended semantics (Appendix A). Developing methods for automated verification or iterative refinement of LLM-generated objectives, potentially incorporating feedback from simulation rollouts [15, 16] or interaction, would significantly enhance the autonomy and reliability of the system. Integrating synergistic environment generation [14] could also improve the alignment between the task, the hierarchy, and the training environment.

Observation Space Abstraction: To manage simulation speed and focus on hierarchical learning aspects, our current implementation utilises privileged state information (e.g., object types, sizes, relative locations, proprioception) rather than raw visual input like camera feeds. While this provides a reasonable proxy for the information obtainable via perception systems, it bypasses the challenges of visual processing and state estimation. Integrating GenHRL with learned visual representations or end-to-end visuomotor policies remains an important next step for real-world applicability.

Simulation-to-Reality Gap: Our evaluations are currently confined to simulation. Transferring the learned hierarchical skills to physical hardware will necessitate bridging the sim-to-real gap, which is out of scope for this paper. While the modular nature of the learned skills may inherently offer robustness benefits, explicitly incorporating techniques like domain randomization [36] during the skill learning phase (Section 3.3) within GenHRL is a crucial direction for future work to ensure robustness against variations in real-world dynamics.

Scope of Skills: The current framework primarily focuses on generating hierarchies composed of sub-goal oriented skills, where success is defined by reaching specific states or configurations. Generating and learning purely behavioural skills (e.g., performing a backflip, continuous juggling) where the objective is trajectory-centric rather than state-centric, proved more challenging for LLM-based objective generation in our setup. Extending GenHRL to effectively handle such behavioural abstractions is an area for further investigation.

Learning Strategy: Training currently proceeds sequentially through the hierarchy. Exploring concurrent training approaches, potentially involving fine-tuning lower-level skills based on the context provided by higher-level skill execution, could further improve sample efficiency and policy coordination within the hierarchy [31].

References

[1] S. Pateria, B. Subagdja, A.-h. Tan, and C. Quek. Hierarchical reinforcement learning: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 54:1–35, 2021.

[2] P. S. Rosenbloom and A. Newell. The chunking of goal hierarchies: A generalized model of practice. *Machine learning-an artificial intelligence approach*, 2:247, 1986.

[3] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial intelligence*, 33(1):1–64, 1987.

[4] E. Brunskill and L. Li. Pac-inspired option discovery in lifelong reinforcement learning. In *International conference on machine learning*, pages 316–324. PMLR, 2014.

[5] J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[6] O. Nachum, S. S. Gu, H. Lee, and S. Levine. Data-efficient hierarchical reinforcement learning. *Advances in neural information processing systems*, 31, 2018.

[7] T. P. Cannon and Ö. Simsek. Accelerating task generalisation with multi-level hierarchical options. *arXiv preprint arXiv:2411.02998*, 2024.

[8] A. Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pages 278–287. Citeseer, 1999.

[9] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine. Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070*, 2018.

[10] A. Sharma, S. Gu, S. Levine, V. Kumar, and K. Hausman. Dynamics-aware unsupervised discovery of skills. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=HJgLZR4KvH.

[11] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022.

[12] J. Wen, J. Guan, H. Wang, W. Wu, and M. Huang. Codeplan: Unlocking reasoning potential in large language models by scaling code-form planning. In *The Thirteenth International Conference on Learning Representations*, 2024.

[13] M. Klissarov, D. Hjelm, A. Toshev, and B. Mazoure. On the modeling capabilities of large language models for sequential decision making. *arXiv preprint arXiv:2410.05656*, 2024.

[14] Y. Wang, Z. Xian, F. Chen, T.-H. Wang, Y. Wang, K. Fragkiadaki, Z. Erickson, D. Held, and C. Gan. Robogen: Towards unleashing infinite data for automated robot learning via generative simulation. *arXiv preprint arXiv:2311.01455*, 2023.

[15] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, L. Fan, and A. Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.

[16] Y. J. Ma, W. Liang, H.-J. Wang, S. Wang, Y. Zhu, L. Fan, O. Bastani, and D. Jayaraman. Dreureka: Language model guided sim-to-real transfer. *arXiv preprint arXiv:2406.01967*, 2024.

[17] M. Klissarov, M. Henaff, R. Raileanu, S. Sodhani, P. Vincent, A. Zhang, P.-L. Bacon, D. Precup, M. C. Machado, and P. D'Oro. Maestromotif: Skill design from artificial intelligence feedback. *arXiv preprint arXiv:2412.08542*, 2024.

[18] R. S. Sutton, A. G. Barto, et al. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[19] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[20] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[21] M. Mittal, C. Yu, Q. Yu, J. Liu, N. Rudin, D. Hoeller, J. L. Yuan, R. Singh, Y. Guo, H. Mazhar, A. Mandlekar, B. Babich, G. State, M. Hutter, and A. Garg. Orbit: A unified simulation framework for interactive robot learning environments. *IEEE Robotics and Automation Letters*, 8(6):3740–3747, 2023. doi:10.1109/LRA.2023.3270034.

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

[23] K. Gregor, D. J. Rezende, and D. Wierstra. Variational intrinsic control. *arXiv preprint arXiv:1611.07507*, 2016.

[24] P. Dayan and G. E. Hinton. Feudal reinforcement learning. *Advances in neural information processing systems*, 5, 1992.

[25] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *International conference on machine learning*, pages 3540–3549. PMLR, 2017.

[26] E. A. McGovern. *Autonomous discovery of temporal abstractions from interaction with an environment*. University of Massachusetts Amherst, 2002.

[27] Ö. Şimşek, A. P. Wolfe, and A. G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd international conference on Machine learning*, pages 816–823, 2005.

[28] P.-L. Bacon, J. Harb, and D. Precup. The option-critic architecture. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

[29] A. Harutyunyan, W. Dabney, D. Borsa, N. Heess, R. Munos, and D. Precup. The termination critic. *arXiv preprint arXiv:1902.09996*, 2019.

[30] M. Riemer, M. Liu, and G. Tesauro. Learning abstract options. *Advances in neural information processing systems*, 31, 2018.

[31] A. Levy, G. Konidaris, R. Platt, and K. Saenko. Learning multi-level hierarchies with hindsight. In *Proceedings of International Conference on Learning Representations*, 2019.

[32] J. B. Evans and Ö. Şimşek. Creating multi-level skill hierarchies in reinforcement learning. *Advances in Neural Information Processing Systems*, 36:48472–48484, 2023.

[33] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

[34] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

[35] J. Andreas, D. Klein, and S. Levine. Modular multitask reinforcement learning with policy sketches. In *International conference on machine learning*, pages 166–175. PMLR, 2017.

[36] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 23–30. IEEE, 2017.

A Human Verifier Changes Made

In Table 2 we detail the minor changes made to the rewards and success functions outputted from the GenHRL by a human verifier. The largest changes occurred in the most dynamic skills such as jumping. Most of the others were removing redundant rewards which slowed training or tuning thresholds produced.

Table 2:

Skill name	**Reward modifcations**	**Success modifications**
WalkToLowWall	Removed unnecessary activation condition for the y shaping reward	Added a too close threshold value of 0.5
PrepareForJumpOverLowWall	Removed unnecessary reward for moving closer to the low wall.	Increased duration for success to 1 second
ExecuteJumpOverLowWall	Added a target Z height of 0.9 instead of maximising Z. Added torch.exp to the X target calculation to keep positive	
LandStablyAfterLowWall	Removed x target, removed activation.	Added a lower bound for pelvis height
JumpOverLowWall	Combined x target and z target for pre-wall conditions Increased the Z target height.	
WalkToLargeSphere		
PositionHandsForPushLargeSphere	Removed Z components of rewards. Removed pelvis position shaping reward	Remove Z components of rewards
PushLargeSphereForward		
EnsureHighWallFalls	Removed or reduced most weights except pelvis stability	Increase duration from 0.5 to 2seconds
PushLargeSphereToHighWall		Added z component of wall fall
WalkToSmallSphere	Changed torch.min to torch.abs	Added pelvis height constraint
ExecuteKickSmallSphereForward		
KickSmallSpherePastBlock		
WalkToBlock	Added Y component to reward	Added Y component to criteria
PrepareForJumpOntoBlock		
ExecuteJumpOntoBlock	Reduced weight of avoiding collision with block to 0 Added Y component to approach reward Changed main reward from average to minimum Added function to promote feet being under pelvis	Added Y component
JumpOntoBlock	Added non-activation default to -2 from 0	