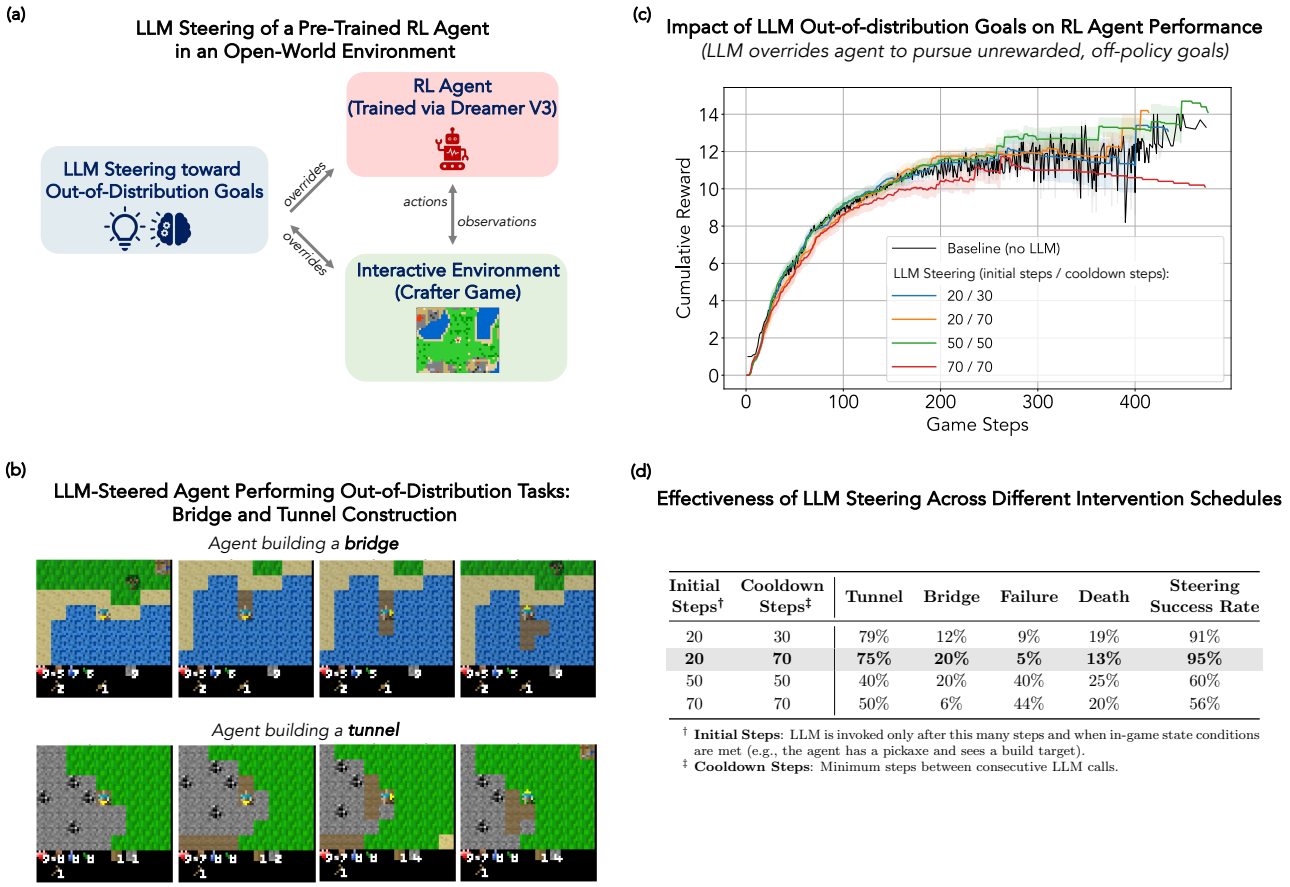
LLM Steering of Pre-Trained RL Agents Toward Out-of-Distribution Goals

Mehdi Soleimanifar
mehdi@caltech.edu

Abstract

We investigate how large language models (LLMs) can expand the objective space of pre-trained reinforcement learning (RL) agents, enabling open-ended behavior in complex environments without the need for retraining. In our setup, a DreamerV3 agent is trained in the Minecraft-inspired Crafter environment using standard rewards based on resource gathering. During inference, an LLM periodically overrides the agent by issuing action sequences aimed at novel, unrewarded, and out-of-distribution objectives–such as building bridges or tunnels. We analyze the trade-offs between the timing and frequency of LLM interventions and their effects on the agent's cumulative reward. Our findings reveal an optimal intervention schedule that balances the pursuit of new, off-policy goals with the preservation of the agent's native exploration and reward-maximizing behaviors.



Initial Steps†	Cooldown Steps‡	Tunnel	Bridge	Failure	Death	Steering Success Rate
20	30	79%	12%	9%	19%	91%
20	**70**	**75%**	**20%**	**5%**	**13%**	**95%**
50	50	40%	20%	40%	25%	60%
70	70	50%	6%	44%	20%	56%

† **Initial Steps**: LLM is invoked only after this many steps and when in-game state conditions are met (e.g., the agent has a pickaxe and sees a build target).
‡ **Cooldown Steps**: Minimum steps between consecutive LLM calls.

Figure 1: **LLM Steering in Crafter.** (a) System overview: a GPT-4.1 policy intermittently overrides a pre-trained Dreamer-V3 agent to pursue unrewarded, off-policy construction goals (bridges, tunnels) in the Crafter environment. (b) Example roll-outs showing successful bridge (top) and tunnel (bottom) construction under LLM control. (c) Cumulative-reward curves for four override schedules. Frequent overrides can divert the agent from its reward-optimized trajectory, potentially lowering cumulative return; sparse schedules mitigate this effect. Shaded bands denote ±1 s.e.m. over five seeds. (d) Outcomes of LLM overrides for four schedules. A 20-step start delay with a 70-step cooldown achieves 95% task success while largely preserving reward, whereas denser schedules increase failure and mortality. Compared to the 30-step cooldown variant,however, the number of constructions is lower.

1 Background

Need for agents. We investigate how large language models can be used to facilitate discovery in interactive environments. On their own, LLMs often struggle to navigate and explore such environments effectively due to limited context, response latency, and inference cost. Instead, an interactive agent can serve as an interface for the LLM, enabling it to explore the environment more effectively. In this setup, the LLM can guide the agent to discover new behaviors, uncover novel strategies, or it can intervene directly to pursue objectives beyond the agent's native capabilities.

Hand-crafted bots vs. RL agents. One common approach involves hand-engineering bots tailored to specific environments. A notable example is the Voyager paper [WXJ+23], which employs the Mineflayer bot to operate within Minecraft. In this framework, the LLM iteratively acquires new capabilities by generating code that instructs the bot to perform increasingly more complex tasks.

Alternatively, general-purpose reinforcement learning (RL) agents can be trained to optimize for predefined goals while exhibiting some exploratory behavior. RL agents are highly effective at maximizing reward within known distributions but typically fall short when it comes to open-ended exploration or the autonomous discovery of novel goals. In contrast, LLMs are well-suited for creative reasoning and goal generation, but lack the fine-grained environmental control RL agents provide.

Steering pre-trained RL with LLMs. In this document, we explore a hybrid framework that leverages the complementary strengths of both systems. An RL agent serves as the primary actor within the environment, while the LLM is intermittently invoked to propose novel goals or strategies that may be outside the RL agent's training distribution. This design aims to balance exploitation and exploration in open-ended, complex environments with large discovery spaces.

Ideally, we would involve the LLM during training—using it to shape rewards and provide dynamic goals—while training the RL agent to respond to LLM guidance without compromising its coherence or exploration capabilities. However, this is computationally demanding. As a more tractable alternative, we experiment with a framework where a pre-trained RL agent serves as the primary controller for environment interaction, while an LLM is selectively invoked to steer the agent toward novel objectives and discoveries beyond the agent's original training scope. This approach addresses the fundamental challenge of balancing exploration and exploitation in environments where the space of possible discoveries is vast and unknown.

2 Experiment Setup

Crafter details. Our experiments use the Crafter environment [Haf21], a 2D, Minecraft-inspired world built around a hierarchy of 22 achievements from basic tasks like chopping wood and crafting tools to advanced milestones such as forging iron and building shelters. These achievements form a directed dependency graph that often requires repeating early sub-tasks (e.g., gathering resources) before unlocking higher-level goals.

The game play shown in Figure 1(b) consists of an image that combines a local top-down view with an on-screen inventory (health, food, water, rest, materials, tools). Episodes terminate upon health dropping to zero—whether from hunger, thirst, exhaustion, hostile attacks, or lava or after 10000 steps. This makes Crafter a rich testbed for resource management, survival, and long-horizon planning.

In Crafter, you earn +1 *only* the first time you unlock each of the 22 achievements, incur small negative penalties whenever you lose health, and receive zero reward for all other actions, so your episode return is the sum of unique-achievement bonuses minus health-loss penalties.

Training. We train an RL agent using the Dreamer V3 [HPBL25] from scratch on an A100 GPU for roughly 5 hours to master Crafter's core objectives: advancing through technological stages (e.g., stone-age to iron-age), collecting essential resources, crafting tools, and fending off enemies. Once trained, the agent reliably unlocks part of achievement chain under its own learned policy.

Inference. As shown in Figure 1(a,b), during inference, we augment the RL agent with a LLM that observes both the agent's visual state and its inventory. The LLM is prompted to generate novel, out-of-distribution objectives, such as constructing bridges or tunneling, that the agent may never have seen during training. This hybrid architecture lets us study how high-level, language-driven planning along with a powerful RL policy can achieve open-ended, emergent behaviors.

LLM-to-agent interface. To translate the LLM's symbolic instructions into concrete actions, we employ a path-finding module that converts each high-level goal into navigable waypoints, and a move interpreter that maps those waypoints into executable game moves. Our path-finding tool and approach here use the setup of [YCS+25] where an LLM controls the entire game with no RL agent involved.

An rough outline of the prompt to the model (gpt-4.1) is as follows:

Prompt outline for bridge/tunnel building

```
You are helping a player navigate in Crafter, a Minecraft-like world. Based on the semantic
analysis below, select one object that would be interesting or useful to move toward.
{semantic_text}
Available objects in view: {objects_list}
1) In one sentence, describe the player's POV, matching the image and map.
2) Then choose 'water' (to build a bridge) or 'stone' (to tunnel). Bridge requires stone; tunnel
requires a pickaxe.
Your current inventory: {current_inventory}
---
Part-2: You've moved next to the object (water or bridge) you have previously chosen.
Here's the updated semantic map: {final_semantic_pov}
Respond with:
A) A one-line scene description.
B) Comma-separated moves (up/down/left/right) to build a p-shape (or its rotation).
```

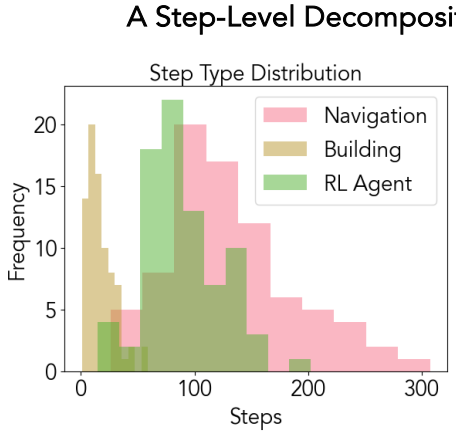
3 Results

When and How Often Should the LLM Intervene? We investigate this question in Figure 1(c, d), where the percentages for successful construction of *Tunnel*, *Bridge*, and *Failure* are conditioned on episodes that include a build attempt. The *Death* rate is computed across all episodes, regardless of build attempts. The overall *Success Rate* indicates the fraction of build attempts that result in either a Tunnel or a Bridge. This analysis reveals several key trends:

- **Early LLM engagement helps.** Runs with an initial intervention delay of 20 steps consistently outperform those with later interventions, achieving higher success rates and greater build yield per LLM call. In our setup, the LLM is permitted to intervene after this initial delay *and* once the agent has acquired the necessary resources (stone or a pickaxe). On average, this resource acquisition occurs at step 20.45.
- **Cooldown controls balance risk and throughput.** A short cooldown (e.g., 20/30) enables more frequent interventions and higher construction activity, but it also increases the risk of agent deaths. This is likely because the Dreamer agent has less time to autonomously gather resources or avoid threats—especially since it does not engage enemies during LLM-guided construction. A longer cooldown (e.g., 20/70) maintains high success rates while reducing risk, striking a better balance between autonomy and LLM assistance.
- **Delayed engagement hampers performance.** Postponing the first LLM intervention until step 50 or 70 results in lower productivity and more frequent failures. By this point, the agent's health, food, water, or rest levels may be depleted, and initiating a build can divert it from critical survival tasks or take it too far from essential resources.

What is the impact of LLM steering on RL Agent's performance? We explore this question in Figure 1(c) where the cumulative reward curves are plotted for baseline and various LLM intervention schedules.

- **Cumulative reward.** LLM steering does not inherently degrade an RL agent's cumulative reward; with a well-designed intervention schedule, performance can closely match that of the baseline. As mentioned earlier, in contrast, interventions that are too sparse or delayed (e.g., 70/70 schedules) tend to disrupt the agent's alignment with its learned reward structure, resulting in a noticeable drop in performance.
- **Variance.** Reward variance remains low where performance is high. Narrow shaded error bands ($\pm$ s.e.m.) for the 50/50 (green) and 20/70 (orange) schedules near the performance plateau indicate consistent outcomes across seeds. In comparison, the 70/70 (red) schedule exhibits broader bands, reflecting greater instability and variability.
- **Average number of steps.** As shown in Figure 2, on average, baseline episodes last 221.12 compared to 239.25 steps in LLM-guided episodes, which includes additional actions related to navigation and bridge/-tunnel construction. This suggests that while the LLM-steered agent spends less time on resource gathering, the strategic interventions by the language model incur only a modest cost in rewards, yet enable richer and more complex behavior.



Metric	Value
Average steps per episode (Baseline)	221.12
Average steps per episode (LLM)	239.25
Mean navigation steps (LLM)	130.8875
Mean construction steps (LLM)	17.6375
Mean RL agent steps (LLM)	90.725

Figure 2: We break down total steps into three categories: *navigation* (traveling to stone or water), *construction* (building a bridge or digging a tunnel), and *RL agent* (pure gameplay under the pre-trained policy). While the overall step count for the LLM-steered and baseline agents remains similar, the LLM-steered agent distributes its effort across all three modes, highlighting its ability to combine high-level planning with low-level execution.

Conclusions. In summary, our results reveal a sweet spot: early LLM engagement followed by infrequent, strategic interventions allows the LLM to guide construction tasks while still giving the agent enough autonomy to explore, gather resources, and manage survival-related needs. Importantly, this approach shows that even without retraining the RL agent or involving LLM calls during training, a pre-trained policy can still be steered toward off-policy, LLM-defined goals. This suggests a generalizable alternative to environment-specific, hand-crafted bots: standard RL training can be paired with LLM guidance at inference time to enable open-ended exploration and novel behaviors.

References

[Haf21] Danijar Hafner. Benchmarking the spectrum of agent capabilities. *arXiv preprint arXiv:2109.06780*, 2021.

[HPBL25] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse control tasks through world models. *Nature*, pages 1–7, 2025.

[WXJ+23] 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. *arXiv preprint arXiv:2305.16291*, 2023.

[YCS+25] Hanqing Yang, Jingdi Chen, Marie Siew, Tania Lorido-Botran, and Carlee Joe-Wong. Llm-powered decentralized generative agents with adaptive hierarchical knowledge graph for cooperative planning. *arXiv preprint arXiv:2502.05453*, 2025.