# REWARD-RATE POLICY GRADIENT FOR EFFICIENT MACHINE LEARNING ENGINEERING AGENTS

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

## ABSTRACT

Traditional reinforcement learning (RL) techniques focus on maximizing expected cumulative reward, where each action assumes to take a constant unit of time. However, this assumption does not hold for agentic RL tasks such as machine learning engineering (MLE) agents, where actions involve data loading, feature engineering, and model training that take variable durations. Efficiency matters in modern agentic RL where actions are costly. To address this limitation, we adapt from continuous-time RL and Semi-Markov Decision Process (SMDP) formulation and propose Reward-rate Policy Gradient (RPG), where we focus on optimizing the *reward rate* — the long-term reward per unit of time. RPG estimates the reward rate from off-policy samples, then charges each action for the time it consumes at that rate. We first conduct theoretical analysis in the bandit setting to establish that RPG approximates the optimal reward rate and empirically demonstrate it outperforms baselines while avoiding enumeration over the policy space, a known issue for an existing method. We then further apply RPG on a small language model (Qwen3.5-4B) with self-improvement loops and empirically show it obtains higher rewards within a fixed time budget than vanilla RL on MLE-Bench and NanoGPT, with a 19.2% and 85.7% margin, respectively. Our method provides a practical solution for optimizing performance under wait time considerations in modern agentic RL tasks, where actions interact with external environments and cost time.

## 1 INTRODUCTION

RL has been used extensively to train language models (LMs) and agents, where the standard approach is to maximize expected cumulative reward (Christiano et al., 2017; Zhang et al., 2025a). Recently, there has been interest in MLE agents, where the goal is to automate the MLE pipeline to achieve best performance on held-out data (Chan et al., 2025; Yang et al., 2025; Nam et al., 2026). Current approaches mostly apply a strong LM in a search procedure: AIDE explores the space of candidate programs as a tree (Jiang et al., 2025), while MLE-STAR retrieves an initial solution with a search engine and then refines its targeted components (Nam et al., 2026). A separate line trains the model itself with RL on performance score (Yang et al., 2025). However, none of these methods perform gradient updates that fundamentally change agents' behavior to *learn* to generate more efficient solutions. In MLE, each action takes variable time and resources, so it is important to consider samples' costs for practicality — we want agents that are not only good, but also *efficient*.

One solution is to use multi-objective RL and search for the Pareto frontier between performance and time (Roijers et al., 2013; Hayes et al., 2022), but computing the set of optimal solutions is often infeasible and compute-heavy (Hayes et al., 2022). These limitations motivate an objective that *intrinsically* aims for long-run efficiency. One natural formulation is the long-run reward per unit of time studied in Semi-Markov Decision Process (SMDP) (Sutton et al., 1999b; Das et al., 1999; György et al., 2007). Intuitively, the method uses RL to maximize relative rewards, calculated from total rewards minus the amount that it would have earned if time were spent optimally. However, performing this computation requires knowing the optimal reward rate *apriori*, which is often not the case in practice, giving a chicken-or-egg dilemma. Existing approaches mainly use two ways to tackle this. Directly optimizing the reward rate with SMDP policy gradient objective (Sutton et al., 1999a) requires estimating the rate of the current policy from on-policy rollouts, an operation that is still expensive when actions are costly. The continuous-time bandit approach (György et al., 2007)



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.

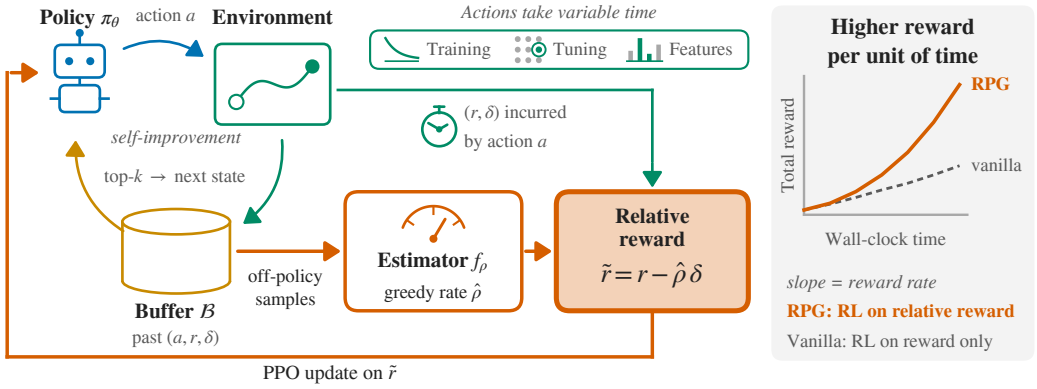



Figure 1: **Overview of RPG in MLE, where our goal is to optimize for reward per unit of time in agentic RL contexts**. Our agent under policy $\pi_\theta$ is asked to self-improve upon its previous MLE scripts from buffer $\mathcal{B}$, and each action $a$ is a proposed script that executes in variable time $\delta$ and admits a performance score $r$ such as AUC. $\hat{\rho}$ is estimated from past samples in buffer $\mathcal{B}$ and captures the cost per unit of time. Relative reward $\tilde{r}$ is then used in PPO to perform RL training. Maximizing the relative reward optimizes for reward rate, giving higher reward per unit of time and more efficient RL agents.

estimates the optimal reward rate from historical samples, but it requires an exhaustive search over the policy space, infeasible in modern agentic RL settings where action space is large.

To that end, we propose Reward-rate Policy Gradient (RPG), which uses off-policy historical samples to estimate the reward rate under the greedy policy and maximizes for relative reward (Figure 1). RPG avoids the limitations of the two previous methods. It does not need time-costly on-policy rollouts for each gradient step, and does not require searching over the policy space. Additionally, RPG is simple to adopt in the current RL pipeline, since it only requires a reward-rate estimator and uses the relative reward as the learning signal. We first study RPG in the bandit setting and theoretically prove that, in expectation, if the greedy policy's reward rate is estimated well, then RPG is equivalent to the Dinkelbach iteration (Dinkelbach, 1967; Schaible, 1976), and natural policy gradient (NPG) (Amari, 1998) is suited for reward-rate optimization due to preconditioning on Fisher information. Experiments on the bandit setting confirm that RPG obtains lower regret than the reward-rate upper-confidence-bound algorithm by György et al. (2007) without the need to search over the policy space. These findings motivate us to apply Proximal Policy Optimization (PPO), a first-order approximation of NPG methods (Kakade, 2001; Schulman et al., 2015), on MLE-Bench and NanoGPT. Compared against vanilla RL baselines on the same time budget, RPG shows a 19.2% average improvement in performance across 20 out of 22 MLE-Bench tasks and an 85.7% margin on NanoGPT. Our findings demonstrate that RPG is a practical solution for future agentic RL frameworks where actions are costly.

## 2 Preliminaries

### 2.1 Semi-MDP Policy Gradient

We introduce the notation and setup for the SMDP setting where each action takes a variable amount of time (Sutton et al., 1999b; Das et al., 1999). Let states be $s \in \mathcal{S}$ and actions be $a \in \mathcal{A}$. At each step $t$, the policy $\pi(a_t \mid s_t)$ is a conditional distribution on the actions. Taking the action $a_t$ gives reward $r_{t+1} := r(s_t, a_t)$ and time $\delta_{t+1} := \delta(s_t, a_t)$, and the transition model is a joint conditional distribution $p(s_{t+1}, \delta_{t+1}, r_{t+1} \mid s_t, a_t)$. Our objective is to maximize the long-term reward per unit of time:

$$\rho(\pi) = \lim_{t\to\infty} \frac{\mathbb{E}_\pi\left[\sum_{j=1}^{t} r_j\right]}{\mathbb{E}_\pi\left[\sum_{j=1}^{t} \delta_j\right]}, \quad \pi^* = \arg\max_\pi \rho(\pi). \tag{1}$$

For a parameterized policy $\pi_\theta$, taking the gradient gives

$$\nabla_\theta \rho(\pi_\theta) = \frac{\sum_s d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(a \mid s) Q^{\pi_\theta}(s,a)}{\mathbb{E}_{\pi_\theta}[\delta]}, \tag{2}$$

where $Q^{\pi_\theta}(s,a) = \mathbb{E}_{\pi_\theta}\big[r - \rho(\pi_\theta)\,\delta + V^{\pi_\theta}(s') \mid s, a\big]$ is the *differential* action-value function (see full derivation in Appendix A.1). Empirically, the division by $\mathbb{E}_{\pi_\theta}[\delta]$ can be dropped since modern adaptive optimizers like Adam (Kingma & Ba, 2015) and gradient clipping reduce its impact on the gradients. Thus, the implementation is a straight-forward plug-in using the *relative reward* $\tilde{r}_t$ into existing RL algorithms

$$\tilde{r}_t := r_t - \hat{\rho}_t\,\delta_t, \qquad e_t = \tilde{r}_t + V_\phi(s_{t+1}) - V_\phi(s_t), \qquad \hat{A}_t = \sum_{l \geq 0} \lambda^l\, e_{t+l}, \tag{3}$$

$$\mathcal{L}(\theta) = \mathbb{E}_t\Big[\min\big\{w_t(\theta)\hat{A}_t,\ \mathrm{clip}\big(w_t(\theta), 1-\epsilon, 1+\epsilon\big)\hat{A}_t\big\}\Big], \qquad w_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}, \tag{4}$$

where $\hat{\rho}_t$ is a Monte Carlo approximation of $\rho(\pi)$ from on-policy rollouts. Intuitively, the relative reward $\tilde{r}_t$ is the gain obtained for the selected action when $\delta_t$ units of time are used at a cost of $\rho(\pi)$, so actions are ranked by their relative performance per unit of time.

### 2.2 Continuous Time Bandit

Reward-rate RL has also been studied in the continuous-time bandit setting (György et al., 2007). The agent observes context $x_t$ drawn i.i.d. from a fixed distribution on a finite set $\mathcal{X}$, and each action $a_t \in \mathcal{A}$ yields a reward $r_{t+1}$ and time $\delta_{t+1}$, with $r \in [r_{\min}, r_{\max}]$ and $\delta \in [\delta_{\min}, \delta_{\max}]$, $\delta_{\min} > 0$. The policy $u \in \mathcal{U} = \{u : \mathcal{X} \to \mathcal{A}\}$ is assumed to be stationary and deterministic. Rewards $\{r_j\}_{j=1}^t$ and durations $\{\delta_j\}_{j=1}^t$ are drawn i.i.d. across steps from a fixed distribution given $(x,a)$. The objective is to maximize the *relative value* under the optimal reward rate:

$$q^*(x,a) \;=\; \mathbb{E}\big[r(x,a)\big] - \rho^*\,\mathbb{E}\big[\delta(x,a)\big], \qquad \rho^* = \max_{u \in \mathcal{U}} \rho(u), \tag{5}$$

so that a policy is optimal when it selects $a^* = \arg\max_{a \in \mathcal{A}} q^*(x,a)$ at every context. When the reward rate is not optimal, $\rho \neq \rho^*$, we denote the relative value as $q_\rho(x,a)$. Regret is thus defined as

$$R_t = \rho^* \sum_{j=1}^t \delta_j - \sum_{j=1}^t r_j. \tag{6}$$

To minimize regret, the algorithm starts by estimating $\rho^*$ with $\hat{\rho}_t$ using the past history $\{(r_j, \delta_j)\}_{j=1}^t$ and performs an exhaustive search over all possible policies

$$\hat{\rho}_t = \max_{u \in \mathcal{U}} \big\{\bar{\rho}_t(u) - b_t(u)\big\}, \qquad \bar{\rho}_t(u) = \frac{\sum_{j=1}^t \mathbb{1}\{a_{j-1} = u(x_{j-1})\}\, r_j}{\sum_{j=1}^t \mathbb{1}\{a_{j-1} = u(x_{j-1})\}\, \delta_j}, \tag{7}$$

which subtracts a confidence width $b_t(u)$, giving a lower bound satisfying $\hat{\rho}_t \leq \rho^*$ with high probability. Actions are then selected optimistically with the estimated relative value, where $\bar{r}_t$ and $\bar{\delta}_t$ are sample averages and $c_t$ is a confidence bonus

$$a_t = \arg\max_{a \in \mathcal{A}} \big\{\bar{r}_t(x_t,a) - \hat{\rho}_t\,\bar{\delta}_t(x_t,a) + c_t(x_t,a)\big\}. \tag{8}$$

This algorithm yields $O(\log T)$ expected regret bound but requires enumerating $|\mathcal{U}| = |\mathcal{A}|^{|\mathcal{X}|}$ policies at each step. More details on the algorithm's construction of $c_t$ and $b_t(u)$ are in Appendix A.2.

## 3 Reward-rate Policy Gradient

In this section, we propose RPG for optimizing the reward rate practically. We provide an outline for RPG in Section 3.1 that shows how we avoid previous algorithms' limitations. Section 3.2 shares our theoretical insights, which were validated through experiments in Section 3.3.

**Algorithm 1** RPG

**Require:** Policy $\pi_\theta$, greedy reward rate estimator $f_\rho$, buffer $\mathcal{B}$, $\hat{\rho}_1 \leftarrow 0$.

▷ *$f_\rho$ is a Normal-Inverse-Wishart (NIW) posterior, but it can also be Robbins-Monro or neural networks.* ◁

**for** $t = 0$ **to** $T - 1$ **do**

  Sample actions $a_t \sim \pi_{\theta_t}(\cdot \mid s_t)$, execute $a_t$ and observe reward $r_{t+1}$ and time $\delta_{t+1}$.

  Store $a_t$, $\delta_{t+1}$ and $r_{t+1}$ in buffer $\mathcal{B}$.

  Fit $f_\rho$ on $\mathcal{B}$, then get $\hat{\rho}_{t+1} = f_\rho(\mathcal{B})$. ▷ *Fit NIW posterior and draw samples.*

  Calculate relative reward $\tilde{r}_{t+1} = r_{t+1} - \hat{\rho}_{t+1}\,\delta_{t+1}$ and advantage $\hat{A}_{t+1}$.

  $\theta_{t+1} \leftarrow \texttt{PPO}(\theta_t, \hat{A}_{t+1})$.

### 3.1 Main Algorithm

**Current limitations.** We first describe limitations of current approaches and then share the intuition behind our algorithm design. The SMDP policy gradient theorem in Equation (2) tells us it is viable to do gradient-based optimization on a parameterized policy to maximize reward rate. However, the core constraint in practice is that it requires multiple rollouts of the on-policy trajectory $\{(r_j, \delta_j)\}_{j=1}^{T}$ to estimate $\rho(\pi)$. Since the time $\delta_j$ for each step is variable and costly, this makes the estimation take a long time, at least $\sum_{j \le T} \delta_j$, for each gradient update, making it impractical for training efficient MLE agents. On the other hand, the continuous-time bandit approach is an alternative by optimizing for the relative reward using an off-policy reward rate estimate $\hat{\rho}_t$ (Equation (7)), but the limitation being that it involves enumeration over all possible policies.

**Proposed algorithm.** Given the limitations discussed above, our design principle for creating a practical reward-rate RL algorithm is as follows. We first use a predictor $f_\rho$ fitted on the historical buffer $\mathcal{B}$ to approximate the reward rate and use the predicted rate $\hat{\rho}$ to calculate relative reward $\tilde{r}$. This design allows us to avoid sampling on-policy long-horizon rollouts. Unlike the continuous-time bandit setting where policies are deterministic (György et al., 2007), LMs have stochastic policies, so we apply policy gradient updates on expected relative reward to train LMs agents. Our algorithm is shown in Algorithm 1. Ideally, the reward-rate predictor $f_\rho$ should approximate the optimal $\rho^*$, which then serves as $\hat{\rho}$ for the relative reward. However, directly estimating $\hat{\rho} \approx \rho^*$ is difficult, even in the bandit setting, since it requires enumeration over the policy space. Thus, to make the approach practical, we set $f_\rho$ to estimate the greedy policy's reward rate. This choice is supported by our results in the bandit setting (Sections 3.2 and 3.3), where we demonstrate that a softmax policy with NPG updates converges to the optimal $\rho^*$ through both theory and experiments. Empirically on MLE-Bench and NanoGPT, we show that PPO also performs well despite it being a first-order approximation of NPG. The estimator $f_\rho$ can be anything that estimates the greedy reward rate, such as Bayesian inference, Robbins-Monro iteration, or neural networks.

### 3.2 Theoretical Analysis

**Setup.** We share our findings in the bandit setting, which provides motivation for approximating the greedy reward rate for $\hat{\rho}$ and using PPO in our algorithm design. Our analysis is focused on the expected setting with a softmax policy $\pi_\theta(a \mid x) = \exp(\theta_{a,x}) / \sum_b \exp(\theta_{b,x})$. Our assumptions are as follows: (1) boundedness in reward and time, (2) positive gap condition for relative value $\min_x \left[q^*(x, a^*(x)) - \max_{a \neq a^*(x)} q^*(x, a)\right] > 0$, and (3) contexts are drawn i.i.d. from $p(x)$. Details of the analysis are in Appendix B.

**Approximate Dinkelbach iteration.** We first analyze under which conditions an arbitrary sequence of reward rates $\{\rho_k\}_{k=1}^{\infty}$ would be near the optimal $\rho^*$. Theorem 1 shows that if $\rho_{k+1}$ is within a asymmetric $\gamma$-tolerance band around the greedy reward rate $\rho(u_{\rho_k})$ at every step $k \ge 2$, then the sequence would be contained within an interval around the optimal reward rate $\rho_k \in [(1 - R\gamma/(1+\gamma))\rho^*, (1+\gamma)\rho^*]$ as $k \to \infty$, which suggests we are approximately doing a Dinkelbach iteration (Dinkelbach, 1967). When the estimation is exact, $\gamma = 0$, the sequence reaches $\rho^*$ in finite steps. Theorem 1 suggests that we can use an estimation of the greedy reward rate as $\hat{\rho}$ for relative reward calculation and iterate greedily to approximate $\rho^*$. This finding holds tremendous value in practice, since predicting the greedy reward rate at the current $\rho_k$ is an easier task than

estimating $\rho^*$, where the latter requires exhaustive search over the policy space but the former can be approximated with historical samples in $\mathcal{B}$ by selecting top solutions according to $\tilde{r}(\rho_k) = r - \rho_k\,\delta$.

**Theorem 1.** *Let $\gamma \in [0,1]$ be a tolerance term, $\rho_1 \in [0, \rho_{\max}]$, and $\chi := 1 - \delta_{\min}/\delta_{\max} \in (0,1)$. For every $k \geq 1$, let $u_{\rho_k}(x) \in \arg\max_a q_{\rho_k}(x,a)$ be a greedy policy under any tie-breaking and $\rho(u_{\rho_k})$ is the reward rate of the greedy policy, let $\tilde{\rho}_{k+1}$ be any value with the sequence satisfying the condition $\rho(u_{\rho_k})/(1+\gamma) \leq \tilde{\rho}_{k+1} \leq (1+\gamma)\,\rho(u_{\rho_k})$, $\rho_{k+1} = \mathrm{clip}_{[0,\rho_{\max}]}(\tilde{\rho}_{k+1})$ where $\mathrm{clip}_{[0,\rho_{\max}]}(z) := \min\{\max\{z,0\}, \rho_{\max}\}$, $\rho_{\max} = r_{\max}/\delta_{\min}$. Then, for every $k \geq 2$,*

$$\rho_k \in \left[\big(1 - R\gamma/(1+\gamma)\big)\rho^* - \big(\chi/(1+\gamma)\big)^{k-2}\rho_{\max},\ (1+\gamma)\rho^*\right], \tag{9}$$

*where $R = \delta_{\max}/\delta_{\min} > 1$. If $\gamma = 0$, then the sequence increases monotonically $\rho_2 \leq \rho_3 \leq \cdots \leq \rho^*$ and the iteration reaches $\rho_k = \rho^*$ after finite steps.*

**NPG and reward-rate RL.** Theorem 1 suggests that if the predicted rate approximates the greedy rate, then it would be contained within an interval around the optimal $\rho^*$, but it does not make any statements about the training dynamics, so we further analyze the behavior of $\pi_\theta$. Define $A_\theta(\rho, x) = (q_\rho(x,\cdot) - \langle \pi_\theta(\cdot \mid x), q_\rho(x,\cdot)\rangle\,\mathbf{1})$, where $\mathbf{1} = [1 \ldots 1] \in \mathbb{R}^{|\mathcal{A}|}$. The gradient update rules are

$$\theta'_x \leftarrow \theta_x + \eta\, A_\theta(\rho, x) \quad \text{(NPG)}, \qquad \theta'_x \leftarrow \theta_x + \eta\, \pi_\theta(\cdot \mid x) \odot A_\theta(\rho, x) \quad \text{(Standard PG)}.$$

We thus hypothesize that NPG is more suited for reward-rate RL since it has less coupling between $\pi_\theta$ and the magnitude of $A_\theta$. Coupling would be undesirable since $\theta_x$ is updated proportionally to the policy weight $\pi_\theta(\cdot \mid x)$, reducing the signal for actions with large advantage $A_\theta$ but low $\pi_\theta(\cdot \mid x)$[1]. We establish the validity of NPG updates in Theorem 2. It shows that the natural gradient updates are equivalent to a Boltzmann policy on the relative value $q_{\bar{\rho}_k}(x,a)$ with a running average rate $\bar{\rho}_k$. If $\rho_1$ is initialized within $[0, \rho^*]$, and greedy reward rate estimation is exact $\rho_{k+1} = \rho(u_{\bar{\rho}_k})$, then the error between $\bar{\rho}_k$ and $\rho^*$ shrinks in $O(k^{-(1-\chi)})$. Furthermore, when $\bar{\rho}_k$ is within $\bar{\varepsilon}$ of $\rho^*$, which Equation (10) guarantees after finitely many steps, the greedy action at $\bar{\rho}_k$ becomes the optimal one, and the policy increases the optimal arm's probability towards 1. This finding suggests that NPG methods bring performance benefits in reward-rate RL and motivates us to apply PPO in practice, a first-order approximation of NPG (Schulman et al., 2017).

**Theorem 2.** *Let $\rho_1, \rho_2, \ldots$ be a sequence of reward rates and let $\theta^{(k)}$ be the softmax logits after $k$ NPG update steps with learning rate $\eta$ from the zero initialization, with $\bar{\rho}_k = \sum_{n \leq k} \rho_n/k$. Then for every $x$ and $k \geq 1$, $\pi_{\theta^{(k)}}$ is the Boltzmann policy of $q_{\bar{\rho}_k}(x,a)$ with inverse temperature $\eta k$. If moreover $\rho_1 \in [0, \rho^*]$ and $\rho_{k+1} = \rho(u_{\bar{\rho}_k})$, then $\bar{\rho}_k$ increases to $\rho^*$ in $O(k^{-(1-\chi)})$*

$$\rho^* - \bar{\rho}_k \ \leq\ (\rho^* - \bar{\rho}_1)\,\exp(1-\chi)\,(k+1)^{-(1-\chi)}. \tag{10}$$

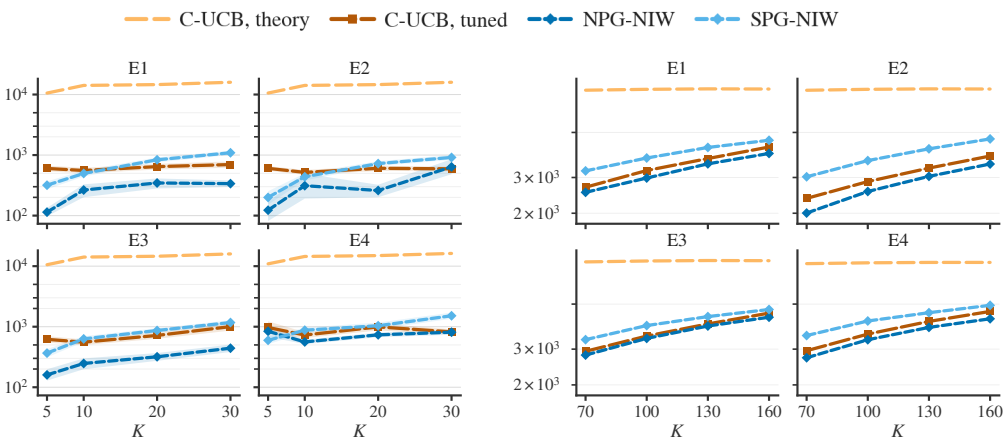
*Furthermore, let $\delta_{\mathrm{spr}} \in (0, \delta_{\max} - \delta_{\min}]$ be the maximum time gap, and $\Delta^*$ be the minimum relative value gap under $\rho^*$, define $\bar{\varepsilon} := \Delta^*/(2\delta_{\mathrm{spr}})$, for every $k \geq k_0$, the greedy policy is optimal*

$$\pi_{\theta^{(k)}}(a^* \mid x) \geq 1 - (|\mathcal{A}|-1)\,\exp\big(-\eta\,\Delta^* k/2\big), \quad k_0 := \left\lceil \big(\exp(1-\chi)\,(\rho^* - \bar{\rho}_1)/\bar{\varepsilon}\big)^{1/(1-\chi)} \right\rceil. \tag{11}$$

### 3.3 Bandit Validation

Our theory suggests that NPG has a decoupling effect on reward-rate optimization, and using greedy reward rate as $\hat{\rho}$ can also approximate the optimal rate as iterations grow. The former brings performance benefits, whereas the latter is a feasibility benefit in large action and context spaces where an exhaustive search over the policy space is intractable. We conduct experiments to examine whether these benefits hold true in practice. Our finding is in Figure 2, and NPG-NIW uses softmax policy with NPG updates and NIW posterior fitted on greedy historical samples for $\hat{\rho}$ estimation. Continuous-time upper confidence bound (C-UCB) is the approach by György et al. (2007). We first observe that if we use C-UCB's constants exactly as in the paper, it breaks since the constants are too large (C-UCB, theory), but treating them as tunable hyperparameters allows it to work (C-UCB, tuned). When only having small context and action spaces (Figure 2a), C-UCB uses exhaustive search for $\hat{\rho}$ estimation. When the context and action space is large (Figure 2b), C-UCB

[1] In standard RL setting, this finding has already been observed Kakade (2001), but in reward-rate RL, $A_\theta$ is a function of $\rho$ which is estimated from policy $\pi_\theta$, so the amount of coupling is stronger.




(a) 4 contexts $|\mathcal{X}| = 4$.

(b) 64 contexts $|\mathcal{X}| = 64$.

Figure 2: **Our proposed approach NPG-NIW obtains similar or lower regret than C-UCB while avoiding exhaustive search over policies**. Standard policy gradient (SPG-NIW) also performs worse than NPG, verifying our theory that NPG's decoupling helps reward-rate RL. y-axis is regret at the final step over 100 trials, x-axis is the number of arms $K$, and 4 environments are considered: (E1) independent reward and time, (E2) time and reward have 0.8 correlation, (E3) time and reward have $-0.8$ correlation, and (E4) reward and time are lognormal with $+0.5$ correlation.

cannot afford to enumerate over all policies, so we use an approximation where 2048 policies are drawn uniformly at random from past visited ones to predict $\hat{\rho}$. Across both settings, we observe that NPG-NIW obtains lower or similar regret as C-UCB baselines, thus confirming our hypothesis that greedy reward rate estimation can be used for $\hat{\rho}$ without enumerating the policy space. When comparing NPG with standard policy gradient updates (SPG-NIW), we observe that it consistently obtains higher regret than NPG, supporting our theoretical finding that NPG provides performance benefits. More experiments and implementation details are in Appendices C.1, D.1 and D.2.

## 4 Reward-rate for MLE Agent

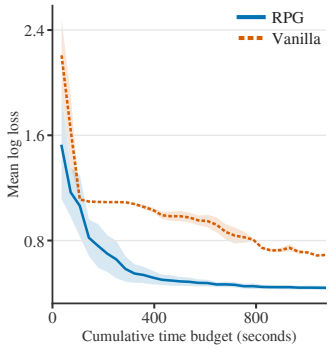
In this section, we present our results demonstrating that RPG finds good solutions in less time than vanilla RL, suggesting that reward-rate RL is a practical solution for training efficient agents. We provide our empirical results on MLE-Bench and NanoGPT in Sections 4.2 and 4.3.

### 4.1 Instantiating Language Models for Reward-rate Policies

**Environment setup.** The theory in Section 3.2 suggests that we should approximate the greedy reward rate for $\hat{\rho}$ and use PPO for training, but training MLE agents also requires deliberate environment setup, which we share in this section. We start with a base prompt as our initial state $s_0$ containing task descriptions such as data loading path, meaning of features, and the performance metric used for the evaluation. We also explicitly ask the LM agent to take efficiency into consideration by encouraging it to use a subset of features. An action $a_0 \sim \pi(\cdot \mid s_0)$ is the generated script by the agent, and the code is executed in a sandbox environment to produce time $\delta_1$ and reward $r_1$. If the script successfully executes, $r_1$ would be a performance score. A timeout budget $\delta_{\max}$ is set as the upper limit of the time allowed for script execution. For scripts that fail to run, such as when they have a bug, we encounter trouble assigning their reward rate since their reward is 0 and $\tilde{r} = -\hat{\rho}\,\delta$ is maximized by $\delta \to 0$. This implies a script that fails on its first line dominates all honest attempts. Thus, we charge every failure the full time limit $\delta_{\max}$ and a constant penalty of $-10$, preventing the policy from reward-hacking by failing fast, and executable scripts have higher rewards than non-executable ones.

Table 1: **RPG vs. vanilla RL on 22 MLE-Bench Lite tasks**, evaluated on mean performance score across self-improving chains within a fixed-time budget. (↑) and (↓) means higher and lower is better, respectively. % improv. is the relative improvement of RPG over vanilla. On average, RPG performs 19.2% better than vanilla, after dropping the highest and lowest numbers.

| Task | RPG | Vanilla | % improv. | Task | RPG | Vanilla | % improv. |
|---|---|---|---|---|---|---|---|
| ranzcr-clip (↑) | **0.089 ± 0.046** | 0.014 ± 0.014 | +530.5 | plant-pathology (↑) | **0.526 ± 0.048** | 0.498 ± 0.047 | +5.6 |
| aptos2019 (↑) | **0.564 ± 0.046** | 0.315 ± 0.120 | +78.9 | pizza (↑) | **0.760 ± 0.008** | 0.728 ± 0.019 | +4.4 |
| mlsp-birds (↑) | **0.333 ± 0.154** | 0.226 ± 0.120 | +46.9 | tabular-2022 (↑) | **0.840 ± 0.013** | 0.806 ± 0.028 | +4.2 |
| siim-isic (↑) | **0.660 ± 0.021** | 0.456 ± 0.112 | +44.8 | whale (↑) | 0.493 ± 0.005 | **0.749 ± 0.091** | −34.2 |
| textnorm-en (↑) | **0.880 ± 0.048** | 0.714 ± 0.068 | +23.2 | leaf-classif (↓) | **0.259 ± 0.063** | 0.564 ± 0.059 | +54.0 |
| textnorm-ru (↑) | **0.889 ± 0.021** | 0.730 ± 0.022 | +21.7 | spooky-author (↓) | **0.440 ± 0.015** | 0.690 ± 0.003 | +36.2 |
| tabular-2021 (↑) | **0.907 ± 0.010** | 0.778 ± 0.046 | +16.5 | nyc-taxi (↓) | **4.309 ± 0.029** | 5.349 ± 0.387 | +19.4 |
| insults (↑) | **0.834 ± 0.025** | 0.761 ± 0.024 | +9.6 | dog-breed (↓) | **4.786 ± 0.001** | 4.924 ± 0.139 | +2.8 |
| jigsaw-toxic (↑) | **0.948 ± 0.009** | 0.868 ± 0.028 | +9.2 | dogs-vs-cats (↓) | **0.688 ± 0.005** | 0.703 ± 0.001 | +2.1 |
| histopathologic (↑) | **0.594 ± 0.107** | 0.554 ± 0.021 | +7.2 | nomad2018 (↓) | **0.064 ± 0.002** | 0.064 ± 0.002 | +0.3 |
| aerial-cactus (↑) | **0.954 ± 0.018** | 0.893 ± 0.010 | +6.8 | denoising-docs (↓) | 0.094 ± 0.002 | **0.085 ± 0.013** | −10.1 |




RPG 5.0s log loss 0.469

```
a = LabelEncoder()
y = a.fit_transform(train_df['author'])
b = CountVectorizer()
X = b.fit_transform(train_df['cleaned_text'])
model = LogisticRegression(solver='lbfgs', max_iter=1000, C…
model.fit(X, y)
```

Vanilla 13s log loss 0.727

```
a = TfidfVectorizer(max_features=15000, ngram_range=(1, 2),…
b = CountVectorizer(analyzer='char', max_features=15000)
X1, X2 = a.fit_transform(X_train_sub), b.fit_transform(X_train_sub)
X = np.hstack([X1.toarray(), X2.toarray()])
model = LGBMClassifier(n_estimators=1200, num_leaves=31, ma…
model.fit(X, y)
```

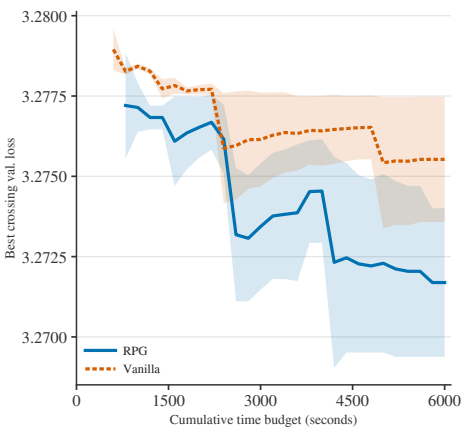
Figure 3: **Example of generated scripts by RPG and vanilla RL**. RPG proposes scripts that use time more efficiently by training on one type of feature with logistic regression, whereas vanilla RL chooses to train on two types of features with 1200 gradient-boosted trees.

**State transition with self-improvement.** Besides setting up the environment and reward properly, we also want our agent to improve based on its own previous attempts, so we would ideally have a self-improving agent making progress on the task over time. We discuss how we connect the agent's own solutions to form self-improvement loops. At every step, we store the observed transition $(a_t, r_{t+1}, \delta_{t+1})$ into the buffer $\mathcal{B}$. The next state $s_{t+1}$ is constructed by concatenating the base prompt $s_0$ with the previous code $a'$ and a self-improvement phrase $i_0$ explicitly asking the agent to improve upon the performance score $r'$ and time $\delta'$. The selection of $(a', r', \delta')$ from buffer $\mathcal{B}$ is based on a top-$k$ selection according to the relative reward $\tilde{r}(\hat{\rho}_{t+1}) = r' - \hat{\rho}_{t+1}\,\delta'$. This naturally gives a self-improvement chain where the agent is tasked with making progress on its best proposed solutions observed thus far.

**Reward-rate predictor.** At each step $t$, our agent uses its history of attempts to predict the reward rate $\hat{\rho}$. The reward-rate estimator $f_\rho$ is a NIW posterior with joint Gaussian likelihood (Murphy, 2012) fitted from historical samples observed in buffer $\mathcal{B}$. We model the pair as $(r_{\text{acc}}, \log \delta_{\text{acc}})$, where $r_{\text{acc}}$ and $\delta_{\text{acc}}$ are the accumulated reward and time, and we take the logarithm because we need time to be positive. We fit the posterior at each step, and sample $M$ reward rates from it and take the 95th percentile $\hat{\rho}_{t+1} = \text{Quantile}_{0.95}(\{\rho_m\}_{m=1}^{M})$. More details are in Appendix C.2.

### 4.2 Evaluation Results on MLE-Bench

We provide our results for MLE-Bench in Table 1. We use Qwen3.5-4B as our LM and apply the method in Section 4.1. Our baseline is vanilla RL, where rewards are performance scores only and time budget is $\delta_{\max}$. We run RPG and the baseline until convergence. Our evaluation metric is the mean reward obtained by the self-improvement chains at every time budget. Specifically, at every step, there are `batch_size` self-improvement chains, and for each chain, a time $\delta$ and reward $r$ are earned. We calculate the mean reward $r$ across chains at every cumulative time budget




Parent 705s val loss 3.2826 misses 3.28

*the orthogonalisation it had*

```
X = X / (X.norm() + 1e-7)
for _ in range(steps):
  A = X @ X.T
  B = b * A + c * A @ A
  X = a * X + B @ X
```

Edit 647s val loss 3.2780 crosses 3.28

*what replaced it*

```
@triton.jit
def scale_kernel(xp, nrm, n, BLOCK):
  x = tl.load(xp + o, mask=mask, other=0.0)
  scale = tl.load(nrm) + 1.0e-7
  tl.store(xp + o, x / scale, mask=mask)
@torch.compile
def ns_iter(X, steps):
normalize(X)
X = ns_iter(X, steps)
def get_lr(it):
  if it < 50:
    return (it + 1) / 50
```

Figure 4: **Budget curve and an example self-improvement step on NanoGPT**. (Left) At the final budget, RPG improves over vanilla by 85.7%. (Right) Example of a self-improvement edit by RPG, which rewrites Newton–Schulz in the Muon optimizer as a triton kernel with a compiled iteration and adds a learning-rate warmup. The new edit crosses the loss target in less time.

$t \in [0, 200, \ldots, T]$ with 200-second intervals. The final time $T$ is decided by taking the cumulative time containing 95% of the self-improvement chains. Each task is evaluated with three repetitions. Across the 22 tasks RPG show an average 19.2% improvement over vanilla at the final time budget, after dropping the task with highest and lowest improvements. The two methods' proposed MLE scripts have distinct differences in wall clock time. An example of the budget curve along with the generated script is shown in Figure 3 for the spooky-author task. Vanilla uses a concatenation of different features and trains a gradient-boosted tree in 13 seconds with log loss 0.727, whereas RPG uses count features along with logistic regression to obtain 0.469 log loss in 5 seconds. The budget curve behind every task is provided in Appendix D.3, and implementation details are in Appendix C.3.

### 4.3 Evaluation Results on NanoGPT

**Motivation and setup.** The MLE-Bench setting discussed above is based on Kaggle competitions. To extend the scope of reward-rate RL agents, we also apply RPG to the NanoGPT speedrun challenge (Karpathy, 2022; Jordan et al., 2024), where the task is to train a tiny LM down to a validation cross-entropy loss of 3.28 on FineWeb (Penedo et al., 2024) with eight GPUs as fast as possible. This task makes the time taken to reach the 3.28 target loss a natural choice to optimize for. Thus, as a baseline, we implement vanilla RL using time to reach the 3.28 target loss as reward, and compare with RPG using relative reward as the learning signal.

**Setup.** Both vanilla and RPG have a linear bonus terms for crossing the 3.28 target loss, $w\,(3.28 - L)$. NanoGPT requires writing tens of thousands of tokens in its code. Thus, to ensure efficient training and focus on making modifications rather than writing executable code, we ask the agent to read prompt $s_t$ and propose high-level delta code changes along with their reasoning for the modification, which is the action $a_t$. A separate, independent, frontier model with deterministic output (zero temperature) serves as the implementer, which reads the action $a_t$ proposed by the agent and implements the changes. More details are in Appendix C.4.

**Results.** We present our results on NanoGPT in Figure 4, where we observe that RPG gives lower validation loss given the same time budget than vanilla RL. At the final time budget, we observe that RPG has an 85.7% margin below 3.28 compared to the baseline. This can be explained by its objective of using the reward rate, which intrinsically optimizes for the loss per unit of time. An example of a self-improvement edit by RPG is shown on the right side of Figure 4, where the agent writes a triton kernel, a compiled iteration, and a learning rate warm-up in its parent script to drive the loss to the target in less time.

## 5 Related work

**Machine learning engineering agents.** Many works currently ask whether an agent can carry out machine learning engineering tasks in an end-to-end fashion (Nam et al., 2026; Tang et al., 2023; Chan et al., 2025; Li et al., 2024; Zhang et al., 2026; Qiang et al., 2026). Similar lines of work also include using agents on data science tasks such as data preprocessing and hyperparameter tuning (Grosnit et al., 2024; Bendinelli et al., 2025; Zhang et al., 2023; Liu et al., 2024). Most agentic approaches in these tasks involve scaffolding a frozen model, such as by searching the solution space as a tree (Jiang et al., 2025), which do not train MLE-specific experience into the model weights. One of the closest works to ours is by Yang et al. (2025), who trained a small LM with RL on MLE-Bench and observed that actions take a variable amount of time, but the focus was on mitigating the time bias favoring faster solutions in asynchronous RL updates. By contrast, our work directly treats time as a cost term in the RL objective using the SMDP reward-rate formulation and trains agents to learn to take efficient actions.

**Reward-rate RL.** We acknowledge that the reward-rate objective itself is classical. The SMDP formulation provides the theoretical framework for our method (Sutton et al., 1999b), and the continuous-time bandit (György et al., 2007) gives the idea of using off-policy estimates for the optimal reward rate. Similar problems also appear in renewal processes with bandit feedback (Cayci et al., 2019) where the objective is to maximize reward within a fixed horizon. A core mathematical toolkit is Dinkelbach iteration for fractional programming (Dinkelbach, 1967; Schaible, 1976), where all of these works aforementioned rely on it to transform a ratio objective into a sequence of linear ones. Budgeted and knapsack bandits (Badanidiyuru et al., 2013; Ding et al., 2013; Xia et al., 2016) also consider actions with costs, but they enforce a hard constraint on a finite budget. In RL for LMs literature, the environment is episodic with actions assumed to have a constant cost (Schulman et al., 2017; DeepSeek-AI et al., 2025). A survey on agentic RL with LMs (Zhang et al., 2025a) also uses the same formulation, with no notion of how long an action can take. Guan et al. (2025) trades off latency against cost, but requires exploring the Pareto frontier in a multi-agent framework. To our knowledge, no existing work applies reward-rate RL in modern LM agents.

**Self-improvement.** Performing self-improvement by asking an agent to revise its own previous attempts is a well-established pattern, such as critiquing its outputs directly (Madaan et al., 2023), reflecting on feedback from the environment (Shinn et al., 2023), training on the attempts that happened to succeed (Zelikman et al., 2022), and rewriting the scaffolding (Zelikman et al., 2023). The idea of self-improvement loops has also been explored in discovery tasks (Yuksekgonul et al., 2026). Closer to our setting, the loop can be driven by execution feedback, where a model repairs its own program from the error message that the program produced (Chen et al., 2023). The loop can also be used on an agent archive to create a tree of diverse and capable agents through evolutionary search (Zhang et al., 2025b). Zeng et al. (2025) proposes using a balance score for the exploration-exploitation trade-off and updating models on samples that are above a score threshold. Our work focuses on using self-improving loops as state transitions to train agents.

## 6 Conclusion

We proposed a practical reward-rate RL algorithm, namely RPG, for training efficient MLE agents using the SMDP formulation, where each action takes variable time, and the objective is long-run reward per unit of time. RPG uses greedy policy's reward rate estimates for the relative reward calculation and directly maximizes it through PPO. It avoids enumeration over all policies and time-costly on-policy rollouts, two limitations for directly applying the SMDP methods in practice. Through theoretical analysis, simulation experiments, and empirical experiments on MLE-Bench and NanoGPT, we demonstrated that RPG produces an agent policy that learns to take actions efficiently and maximizes reward. There are some limitations in this work. The scope of the empirical experiments is focused on MLE tasks, but other agentic RL tasks where actions cost variable time exist, so it would be interesting to see how reward-rate RL does on these tasks. Theoretically, our analysis focuses on the population-level setting in expectation. A finite-sample analysis could provide further insight.

## AI USE STATEMENT

In this work, we used generative AI tools for: implement methods, support qualitative and thematic data analysis, process experiment data, assist in the writing of proofs, and design or provide feedback on research methodology or experiments. We have not used generative AI tools for: generate synthetic data sets, help develop theoretical models or conceptual frameworks, provide critical ingredients for proving mathematical claims, formulate mathematical claims, propose or refine hypotheses, assist with translation, clean and reformat dataset, interpret results. Additionally, we used generative AI tools for: create or modify scientific figures or images, create or edit software code, creation of artifacts, draft parts of a research paper, summarize or analyse existing literature. We have reviewed all AI-assisted work. We have checked through all the code edited by AI, and all experiments involving AI were verified. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

## REFERENCES

Shun-ichi Amari. Natural Gradient Works Efficiently in Learning. *Neural Computation*, 10(2): 251–276, 1998.

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2):235–256, 2002. doi: 10.1023/A:1013689704352.

Ashwinkumar Badanidiyuru, Robert Kleinberg, and Aleksandrs Slivkins. Bandits with Knapsacks. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pp. 207–216, October 2013. doi: 10.1109/FOCS.2013.30. URL http://arxiv.org/abs/1305.2545. arXiv:1305.2545 [cs].

Tommaso Bendinelli, Artur Dox, and Christian Holz. Exploring llm agents for cleaning tabular machine learning datasets. *arXiv preprint arXiv:2503.06664*, 2025.

Semih Cayci, Atilla Eryilmaz, and R. Srikant. Learning to Control Renewal Processes with Bandit Feedback. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2): 1–32, June 2019. ISSN 2476-1249. doi: 10.1145/3341617.3326158. URL https://dl.acm.org/doi/10.1145/3341617.3326158.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. Mle-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations*, volume 2025, pp. 50466–50494, 2025.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching Large Language Models to Self-Debug, 2023. arXiv:2304.05128.

Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep Reinforcement Learning from Human Preferences. In *Advances in Neural Information Processing Systems*, 2017. arXiv:1706.03741.

Tapas K. Das, Abhijit Gosavi, Sridhar Mahadevan, and Nicholas Marchalleck. Solving Semi-Markov Decision Problems Using Average Reward Reinforcement Learning. *Management Science*, 45(4):560–574, 1999. ISSN 0025-1909. URL https://www.jstor.org/stable/2634824.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang,

Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *Nature*, 645(8081):633–638, September 2025. ISSN 0028-0836, 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://arxiv.org/abs/2501.12948. arXiv:2501.12948 [cs].

Wenkui Ding, Tao Qin, Xu-Dong Zhang, and Tie-Yan Liu. Multi-Armed Bandit with Budget Constraint and Variable Costs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 27(1):232–238, June 2013. ISSN 2374-3468. doi: 10.1609/aaai.v27i1.8637. URL https://ojs.aaai.org/index.php/AAAI/article/view/8637.

Werner Dinkelbach. On Nonlinear Fractional Programming. *Management Science*, 13(7):492–498, 1967. ISSN 0025-1909. URL https://www.jstor.org/stable/2627691.

Antoine Grosnit, Alexandre Maraval, James Doran, Giuseppe Paolo, Albert Thomas, Refinath Shahul Hameed Nabeezath Beevi, Jonas Gonzalez, Khyati Khandelwal, Ignacio Iacobacci, Abdelhakim Benechehab, et al. Large language models orchestrating structured reasoning achieve kaggle grandmaster level. *arXiv preprint arXiv:2411.03562*, 3(4), 2024.

Yilin Guan, Qingfeng Lan, Sun Fei, Dujian Ding, Devang Acharya, Chi Wang, William Yang Wang, and Wenyue Hua. Dynamic Speculative Agent Planning, September 2025. URL http://arxiv.org/abs/2509.01920. arXiv:2509.01920 [cs.AI].

András György, Levente Kocsis, Ivett Szabó, and Csaba Szepesvári. Continuous Time Associative Bandit Problems. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pp. 830–835, 2007.

Conor F. Hayes, Roxana Rădulescu, Eugenio Bargiacchi, Johan Källström, Matthew Macfarlane, Mathieu Reymond, Timothy Verstraeten, Luisa M. Zintgraf, Richard Dazeley, Fredrik Heintz, Enda Howley, Athirai A. Irissappane, Patrick Mannion, Ann Nowé, Gabriel Ramos, Marcello Restelli, Peter Vamplew, and Diederik M. Roijers. A practical guide to multi-objective reinforcement learning and planning. *Autonomous Agents and Multi-Agent Systems*, 36(1):26, April 2022. ISSN 1387-2532, 1573-7454. doi: 10.1007/s10458-022-09552-y. URL http://arxiv.org/abs/2103.09568.

R. Jagannathan. On Some Properties of Programming Problems in Parametric form Pertaining to Fractional Programming. *Management Science*, 12(7):609–615, 1966. doi: 10.1287/mnsc.12.7.609.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. AIDE: AI-Driven Exploration in the Space of Code, 2025. arXiv:2502.13138.

Keller Jordan, Jeremy Bernstein, Brendan Rappazzo, and others. Modded-NanoGPT: Speedrunning the NanoGPT baseline. https://github.com/KellerJordan/modded-nanogpt, 2024. GitHub repository.

Sham M. Kakade. A Natural Policy Gradient. In *Advances in Neural Information Processing Systems*, volume 14, 2001.

Andrej Karpathy. nanoGPT: The simplest, fastest repository for training/finetuning medium-sized GPTs. https://github.com/karpathy/nanoGPT, 2022. GitHub repository.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations*, 2015. arXiv:1412.6980.

Ziming Li, Qianbo Zang, David Ma, Jiawei Guo, Tuney Zheng, Minghao Liu, Xinyao Niu, Yue Wang, Jian Yang, Jiaheng Liu, et al. Autokaggle: A multi-agent framework for autonomous data science competitions. *arXiv preprint arXiv:2410.20424*, 2024.

Siyi Liu, Chen Gao, and Yong Li. Large language model agent for hyper-parameter optimization. *arXiv preprint arXiv:2402.01881*, 2024.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback, 2023. arXiv:2303.17651.

Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Jinwoo Shin, Sercan Arik, and Tomas Pfister. Mle-star: Machine learning engineering agent via search and targeted refinement. *Advances in Neural Information Processing Systems*, 38:116692–116712, 2026.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale, 2024. arXiv:2406.17557.

Rushi Qiang, Yuchen Zhuang, Yinghao Li, Dingu Sagar VK, Rongzhi Zhang, Changhao Li, Ian Wong, Sherry Yang, Percy Liang, Chao Zhang, et al. Mle-dojo: Interactive environments for empowering llm agents in machine learning engineering. *Advances in Neural Information Processing Systems*, 38, 2026.

Diederik Marijn Roijers, Peter Vamplew, Shimon Whiteson, and Richard Dazeley. A Survey of Multi-Objective Sequential Decision-Making. *Journal of Artificial Intelligence Research*, 48: 67–113, 2013. arXiv:1402.0590.

Siegfried Schaible. Fractional Programming. II, on Dinkelbach's Algorithm. *Management Science*, 22(8):868–873, 1976. ISSN 0025-1909. URL https://www.jstor.org/stable/2630018.

John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust Region Policy Optimization. In Francis Bach and David Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 1889–1897. PMLR, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, August 2017. URL http://arxiv.org/abs/1707.06347. arXiv:1707.06347 [cs].

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning, 2023. arXiv:2303.11366.

Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999a. URL https://proceedings.neurips.cc/paper_files/paper/1999/hash/464d828b85b0bed98e80ade0a5c43b0f-Abstract.html.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, August 1999b. ISSN 00043702. doi: 10.1016/S0004-3702(99)00052-1. URL https://linkinghub.elsevier.com/retrieve/pii/S0004370299000521.

Xiangru Tang, Yuliang Liu, Zefan Cai, Yanjun Shao, Junjie Lu, Yichi Zhang, Zexuan Deng, Helan Hu, Kaikai An, Ruijun Huang, et al. Ml-bench: Evaluating large language models and agents for machine learning tasks on repository-level code. *arXiv preprint arXiv:2311.09835*, 2023.

Yingce Xia, Wenkui Ding, Xu-Dong Zhang, Nenghai Yu, and Tao Qin. Budgeted Bandit Problems with Continuous Random Costs. In Geoffrey Holmes and Tie-Yan Liu (eds.), *Asian Conference on Machine Learning*, volume 45 of *Proceedings of Machine Learning Research*, pp. 317–332, Hong Kong, 2016. PMLR.

Sherry Yang, Joy He-Yueya, and Percy Liang. Reinforcement Learning for Machine Learning Engineering Agents, September 2025. URL http://arxiv.org/abs/2509.01684. arXiv:2509.01684 [cs].

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to Discover at Test Time, 2026. URL https://arxiv.org/abs/2601.16175. Version Number: 2.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping Reasoning With Reasoning, 2022. arXiv:2203.14465.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-Taught Optimizer (STOP): Recursively Self-Improving Code Generation, 2023. arXiv:2310.02304.

Weihao Zeng, Yuzhen Huang, Lulu Zhao, Yijun Wang, Zifei Shan, and Junxian He. B-STaR: Monitoring and balancing exploration and exploitation in self-taught reasoners. In *International Conference on Learning Representations*, 2025.

Guibin Zhang, Hejia Geng, Xiaohang Yu, Zhenfei Yin, Zaibin Zhang, Zelin Tan, Heng Zhou, Zhongzhi Li, Xiangyuan Xue, Yijiang Li, et al. The landscape of agentic reinforcement learning for llms: A survey. *arXiv preprint arXiv:2509.02547*, 2025a.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents, 2025b. arXiv:2505.22954.

Shujian Zhang, Chengyue Gong, Lemeng Wu, Xingchao Liu, and Mingyuan Zhou. Automl-gpt: Automatic machine learning with gpt. *arXiv preprint arXiv:2305.02499*, 2023.

Yunxiang Zhang, Muhammad Khalifa, Shitanshu Bhushan, Grant Murphy, Lajanugen Logeswaran, Jaekyeom Kim, Moontae Lee, Honglak Lee, and Lu Wang. Mlrc-bench: Can language agents solve machine learning research challenges? *Advances in Neural Information Processing Systems*, 38, 2026.

# APPENDIX


**A Derivation** **15**

A.1 Semi-MDP Policy Gradient Theorem . . . . . 15

A.2 More on György et al. (2007) . . . . . 16

**B Theoretical Analysis** **17**

B.1 Preliminaries . . . . . 17

B.2 Supporting Lemmas . . . . . 18

B.3 Main Results . . . . . 19

**C Implementation** **25**

C.1 Bandit Experiments . . . . . 25

C.1.1 Bandit Baselines . . . . . 25

C.1.2 Experiment Setup . . . . . 26

C.2 Reward-rate Predictor . . . . . 28

C.3 MLE-Bench . . . . . 28

C.3.1 Environment, Reward and Self-improvement . . . . . 28

C.3.2 Prompts . . . . . 29

C.3.3 Hyperparameters . . . . . 30

C.4 NanoGPT . . . . . 32

C.4.1 Environment and Grader . . . . . 32

C.4.2 Delta Proposals and the Implementer . . . . . 32

C.4.3 Reward . . . . . 32

C.4.4 Prompts and an Example Action . . . . . 33

C.4.5 Hyperparameters . . . . . 37

**D Experiments** **38**

D.1 Bandit Experiments, $|\mathcal{X}| = 4$ . . . . . 38

D.1.1 Regret Curves . . . . . 38

D.1.2 Hyperparameter Selection . . . . . 38

D.2 Bandit Experiments, $|\mathcal{X}| > 4$ . . . . . 42

D.2.1 Regret at the Horizon . . . . . 42

D.2.2 Regret Curves . . . . . 42

D.2.3 Hyperparameter Selection . . . . . 42

D.3 MLE Agent . . . . . 47

# A Derivation

## A.1 Semi-MDP Policy Gradient Theorem

**Setup and notation.** We consider a finite SMDP (Sutton et al., 1999b; Das et al., 1999) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, \delta_{\min}, \delta_{\max})$ with finite state space $\mathcal{S}$ and finite action space $\mathcal{A}$. At each *decision epoch* the agent observes a state $s \in \mathcal{S}$, selects an action $a \in \mathcal{A}$, and the environment returns a triple $(r, \delta, s')$ drawn from the transition kernel

$$p(s', r, \delta \mid s, a), \qquad p(s' \mid s, a) \;=\; \iint p(s', r, \delta \mid s, a)\, \mathrm{d}r \mathrm{d}\delta, \tag{12}$$

where $r \in [r_{\min}, r_{\max}]$ is the reward accumulated over the transition and $\delta \in [\delta_{\min}, \delta_{\max}]$ is its time (or duration). We write $\bar{r}(s,a) := \mathbb{E}[r \mid s, a]$ and $\bar{\delta}(s,a) := \mathbb{E}[\delta \mid s, a]$ for the conditional means under Equation (12). Crucially, the kernel does not depend on the policy parameters, so $\nabla_\theta \bar{r}(s,a) = 0$ and $\nabla_\theta \bar{\delta}(s,a) = 0$.

Let $\pi_\theta(a \mid s)$ be a differentiable stochastic policy parameterized by $\theta \in \mathbb{R}^d$. We drop the subscript $\theta$ for brevity when clear from context. We abbreviate, for a fixed policy $\pi$,

$$\bar{\delta}^\pi(s) \;:=\; \mathbb{E}_\pi[\delta \mid s] \;=\; \sum_a \pi(a \mid s)\, \bar{\delta}(s,a). \tag{13}$$

Let $p^\pi(s' \mid s) := \sum_a \pi(a \mid s)\, p(s' \mid s, a)$ denote the transition matrix of the Markov chain. We write $d^\pi$ for its unique stationary distribution,

$$d^\pi(s') \;=\; \sum_s d^\pi(s) \sum_a \pi(a \mid s)\, p(s' \mid s, a) \qquad \forall\, s' \in \mathcal{S}. \tag{14}$$

We assume that

- $0 < \delta_{\min} \le \delta \le \delta_{\max} < \infty$.
- $|r| \le \max\{|r_{\min}|, |r_{\max}|\} < \infty$.
- $\mathcal{S}$ and $\mathcal{A}$ are finite.

Under these assumptions, the long-run *reward rate* (gain per unit time) of $\pi$ is well defined and independent of the initial state:

$$\rho(\pi) \;:=\; \lim_{T\to\infty} \frac{\mathbb{E}_\pi\big[\sum_{t=1}^T r_t\big]}{\mathbb{E}_\pi\big[\sum_{t=1}^T \delta_t\big]} \;=\; \frac{\mathbb{E}_\pi[r]}{\mathbb{E}_\pi[\delta]}, \qquad \mathbb{E}_\pi[\delta] \;:=\; \sum_s d^\pi(s)\, \bar{\delta}^\pi(s), \tag{15}$$

with $\mathbb{E}_\pi[r] := \sum_s d^\pi(s) \sum_a \pi(a \mid s) \bar{r}(s,a)$.

The differential state- and action-value functions of $\pi$ are

$$V^\pi(s) = \mathbb{E}_\pi\big[\, r - \rho(\pi)\delta + V^\pi(s') \mid s \,\big], \qquad Q^\pi(s,a) = \mathbb{E}\big[\, r - \rho(\pi)\delta + V^\pi(s') \mid s, a \,\big], \tag{16}$$

so that $V^\pi(s) = \sum_a \pi(a \mid s) Q^\pi(s,a)$.

**Proof for the policy gradient.**

$$\nabla_\theta\, \rho(\pi_\theta) = \frac{\sum_s d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(a \mid s) Q^{\pi_\theta}(s,a)}{\mathbb{E}_{\pi_\theta}[\delta]}, \tag{17}$$

where

$$V^\pi(s) = \mathbb{E}_\pi\big[r - \rho(\pi)\delta + V^\pi(s') \mid s\big], \quad Q^\pi(s,a) = \mathbb{E}_\pi\big[r - \rho(\pi)\delta + V^\pi(s') \mid s, a\big].$$

*Proof.* For simplicity, we drop $\theta$ when clear from context; all gradients are with respect to $\theta$. Differentiating $V^\pi(s) = \sum_a \pi(a \mid s) Q^\pi(s,a)$ and applying the product rule,

$$\nabla V^\pi(s) = \sum_a \Big[\nabla \pi(a \mid s)\, Q^\pi(s,a) + \pi(a \mid s)\, \nabla Q^\pi(s,a)\Big]. \tag{18}$$

By definition, $Q^\pi(s,a) = \bar{r}(s,a) - \rho(\pi)\bar{\delta}(s,a) + \sum_{s'} p(s' \mid s,a) V^\pi(s')$. Since the transition kernel does not depend on $\theta$, we have $\nabla \bar{r}(s,a) = 0$ and $\nabla \bar{\delta}(s,a) = 0$, hence

$$\nabla Q^\pi(s,a) \;=\; -\nabla\rho(\pi)\,\bar{\delta}(s,a) \;+\; \sum_{s'} p(s' \mid s,a)\,\nabla V^\pi(s'). \tag{19}$$

Substituting Equation (19) into Equation (18) and using Equation (13),

$$\nabla V^\pi(s) = \sum_a \Big[\nabla\pi(a \mid s) Q^\pi(s,a) + \pi(a \mid s) \sum_{s'} p(s' \mid s,a) \nabla V^\pi(s')\Big] \;-\; \nabla\rho(\pi)\,\bar{\delta}^\pi(s). \tag{20}$$

Multiplying Equation (20) by $d^\pi(s)$ and summing over $s \in \mathcal{S}$,

$$\begin{aligned}\sum_s d^\pi(s) \nabla V^\pi(s) = \sum_s d^\pi(s) \sum_a \nabla\pi(a \mid s) Q^\pi(s,a) \\ + \underbrace{\sum_s d^\pi(s) \sum_a \pi(a \mid s) \sum_{s'} p(s' \mid s,a) \nabla V^\pi(s')}_{(\star)} \;-\; \nabla\rho(\pi) \sum_s d^\pi(s)\,\bar{\delta}^\pi(s).\end{aligned} \tag{21}$$

Exchanging the order of summation in $(\star)$ and using Equation (14) gives,

$$(\star) \;=\; \sum_{s'} \Big[\sum_s d^\pi(s) \sum_a \pi(a \mid s) p(s' \mid s,a)\Big] \nabla V^\pi(s') \;=\; \sum_{s'} d^\pi(s') \nabla V^\pi(s'). \tag{22}$$

The left-hand side of Equation (21) and the term Equation (22) are the same finite quantity, they therefore cancel, leaving

$$0 \;=\; \sum_s d^\pi(s) \sum_a \nabla\pi(a \mid s) Q^\pi(s,a) \;-\; \nabla\rho(\pi)\,\mathbb{E}_\pi[\delta], \tag{23}$$

where we used $\sum_s d^\pi(s)\bar{\delta}^\pi(s) = \mathbb{E}_\pi[\delta]$ from Equation (15). Since $\mathbb{E}_\pi[\delta] \geq \delta_{\min} > 0$ by the boundeness assumption, we may divide by it, which completes the proof. □

### A.2 More on György et al. (2007)

We give the explicit form of the two confidence terms $b_t(u)$ and $c_t(a,s)$ appearing in Equations (7) and (8) (György et al., 2007).

Let

$$T_u(t) = \sum_{j=1}^{t} \mathbb{1}\{a_j = u(s_j)\}, \tag{24}$$

be the number of rounds on which the played action agreed with policy $u$, and

$$T_a(s,t) = \sum_{j=1}^{t} \mathbb{1}\{a_j = a,\, s_j = s\}, \tag{25}$$

be the number of times action $a$ was played in context $s$. Both widths are built from the common scale factor

$$\omega_{t,n} \;=\; \sqrt{\frac{\log\left(t\sqrt{|\mathcal{U}|+1}\right)}{n}}, \qquad |\mathcal{U}| = |\mathcal{A}|^{|\mathcal{S}|}. \tag{26}$$

The width on the rate estimate is

$$b_t(u) \;= \omega_{t,\,T_u(t)}\,\sqrt{2\kappa}, \qquad \kappa \;=\; 2\max\left\{\frac{(r_{\max}-r_{\min})^2}{\delta_{\min}^2},\; \frac{r_{\max}^2\,(\delta_{\max}-\delta_{\min})^2}{\delta_{\min}^4}\right\}, \tag{27}$$

and the width on the action index is

$$c_t(a,s) \;=\; (\alpha_0+\alpha_1)\,\omega_{t,\,T_a(s,t)}, \qquad \alpha_1 = \delta_{\max}\sqrt{2\kappa}, \tag{28}$$

$$\alpha_0 = \sqrt{8\max\left\{(r_{\max}-r_{\min})^2,\; \frac{r_{\max}^2\,(\delta_{\max}-\delta_{\min})^2}{\delta_{\min}^2}\right\}}. \tag{29}$$

The two widths are indexed differently: $b_t(u)$ shrinks in the number of rounds consistent with policy $u$, while $c_t(a,s)$ shrinks in the number of pulls of action $a$ in context $s$. The former is subtracted and the latter added, since $\hat{\rho}_t$ enters Equation (8) with a negative sign; pessimism about $\rho^*$ is optimism about $q^*$.

# B Theoretical Analysis

## B.1 Preliminaries

**Setup and notation.** We use the same notation as other parts of our paper, which are discussed in Section 2. Our setting is same as the bandit setting in Section 2.2.

**Reward rate and value.** A stochastic policy $\pi$ has reward rate

$$\rho(\pi) \;:=\; \frac{\mathbb{E}_p\big[\sum_a \pi(a \mid x)\,\bar{r}(x,a)\big]}{\mathbb{E}_p\big[\sum_a \pi(a \mid x)\,\bar{\delta}(x,a)\big]}, \qquad \rho^* \;:=\; \max_{u \in \mathcal{U}} \rho(u),$$

where the maximum runs over the deterministic policies $\mathcal{U} = \{u : \mathcal{X} \to \mathcal{A}\}$ of Equation (5). For $\rho \in \mathbb{R}$ we define the relative values

$$q_\rho(x,a) \;:=\; \bar{r}(x,a) - \rho\,\bar{\delta}(x,a), \qquad M_\rho(x) \;:=\; \max_a q_\rho(x,a), \qquad F(\rho) \;:=\; \mathbb{E}_p\big[M_\rho(x)\big],$$

with $q^* := q_{\rho^*}$ and $a^*(x) := \arg\max_a q^*(x,a)$.

**Constants.** Since $r_{\min} \geq 0$, every policy's reward rate is in $[0, \rho_{\max}]$, where

$$\rho_{\max} := r_{\max}/\delta_{\min},$$

we define the contraction constant as

$$\chi := 1 - \delta_{\min}/\delta_{\max} \in [0,1),$$

the *time spread*[2] is

$$\delta_{\text{spr}} := \max_x \big[\max_a \bar{\delta}(x,a) - \min_a \bar{\delta}(x,a)\big] \in (0, \delta_{\max} - \delta_{\min}],$$

for a rate $\rho \in \mathbb{R}$, a greedy policy is denoted as

$$u_\rho(x) \in \arg\max_a q_\rho(x,a),$$

thus $u_\rho(x)$ is a greedy action given reward rate $\rho$ in context $x$, under any fixed tie-breaking. Similarly, we define the *greedy margin* at $\rho$ as

$$\Delta(\rho) \;:=\; \min_x \big[\max_a q_\rho(x,a) - \max_{a \neq u_\rho(x)} q_\rho(x,a)\big],$$

it is clear that $\Delta(\rho) \geq 0$ generally, and $\Delta(\rho) > 0$ if and only if the greedy action is unique at every context $x$. With $\Delta(\rho^*)$, we have the gap at the optimal rate

$$\Delta^* \;:=\; \Delta(\rho^*) \;=\; \min_x \big[q^*(x,a^*(x)) - \max_{a \neq a^*(x)} q^*(x,a)\big].$$

We define the error parameter as $\bar{\varepsilon} := \Delta^*/(2\delta_{\text{spr}})$, since $\delta_{\text{spr}} < \delta_{\max}$, we have $\bar{\varepsilon} > \Delta^*/(2\delta_{\max})$.

**Algorithm.** The algorithm is softmax policy with NPG, which keeps a conditional distribution on the actions per context. See Appendix C.1.1 for more details.

**Assumption 1** (A1)**.** Reward and time are bounded with $r \in [r_{\min}, r_{\max}]$, $\delta \in [\delta_{\min}, \delta_{\max}]$, $r_{\min} \geq 0$, $\delta_{\min} > 0$.

**Assumption 2** (A2)**.** $\Delta(\rho^*) = \Delta^* > 0$, which is the positive-gap condition (Auer et al., 2002) for the relative values. Equivalently, for every $x$, the maximizer $a^*(x)$ of $q^*(x,\cdot)$ is unique and the gap between the best and second-best relative value is at least $\Delta^*$ at every context.

**Assumption 3** (A3)**.** Contexts are i.i.d. from $p(x)$, as described in Section 2.2 and same as in György et al. (2007).

[2] We assume the expected time for every $(x,a)$ pair cannot be the same since it would reduce the problem into reward maximization in contextual bandit, thus in the SMDP setting we have $\delta_{\text{spr}} \neq 0$. We do not count it towards our list of assumptions since it is an intrinsic part of the problem formulation.

**Scope.** Our analysis focuses on the algorithm's behavior in expectation, i.e., $\hat{Q}_{x,a}(\rho) = q_\rho(x,a)$.

### B.2 Supporting Lemmas

**Lemma 1.** *For $\rho < \rho'$,*

$$\delta_{\min} \;\le\; \big(F(\rho) - F(\rho')\big)/(\rho' - \rho) \;\le\; \delta_{\max};$$

*in particular $F$ is strictly decreasing.*

*Proof.* Let $\rho < \rho'$, we have

$$q_\rho(x,a) - q_{\rho'}(x,a) = \big(\bar{r}(x,a) - \rho\,\bar{\delta}(x,a)\big) - \big(\bar{r}(x,a) - \rho'\,\bar{\delta}(x,a)\big) = (\rho' - \rho)\,\bar{\delta}(x,a),$$

let $a_\rho = \arg\max_a q_\rho(x,a)$, $a_{\rho'} = \arg\max_a q_{\rho'}(x,a)$, then

$$M_\rho(x) = q_\rho(x,a_\rho) = q_{\rho'}(x,a_\rho) + (\rho' - \rho)\bar{\delta}(x,a_\rho) \le M_{\rho'}(x) + (\rho' - \rho)\delta_{\max},$$

similarly

$$M_\rho(x) \ge q_\rho(x,a_{\rho'}) = q_{\rho'}(x,a_{\rho'}) + (\rho' - \rho)\bar{\delta}(x,a_{\rho'}) \ge M_{\rho'}(x) + (\rho' - \rho)\delta_{\min}.$$

Rearrange, and average over every context $x$ then gives

$$(\rho' - \rho)\delta_{\min} \le F(\rho) - F(\rho') \le (\rho' - \rho)\delta_{\max}.$$

□

**Lemma 2.** *$F(\rho^*) = 0$, so $\rho^*$ is the unique root of $F$; $\rho(\pi) \le \rho^*$ for every stochastic policy $\pi$.*

*Proof.* $F(\rho^*) = 0$ can be established by Bellman optimality equation, and $F$ is strictly decreasing, so $\rho^*$ is unique. For arbitrary policy $\pi$, we have

$$\big(\rho(\pi) - \rho^*\big)D(\pi) = \mathbb{E}_p\left[\sum_a \pi(a \mid x) q^*(x,a)\right] \le \mathbb{E}_p\left[\sum_a \pi(a \mid x) M_{\rho^*}(x)\right] = F(\rho^*) = 0,$$

where $D(\pi) = \mathbb{E}_p\left[\sum_a \pi(a \mid x)\bar{\delta}(x,a)\right]$, thus $\rho(\pi) \le \rho^*$. □

**Lemma 3.** *Let $u_\rho(x) \in \arg\max_a q_\rho(x,a)$. If $\rho \le \rho^*$, then $\rho \le \rho(u_\rho) \le \rho^*$ and*

$$\rho^* - \rho(u_\rho) \;\le\; \chi\,(\rho^* - \rho).$$

*Proof.* Let $u_\rho(x) = \arg\max_a q_\rho(x,a)$ be the greedy policy. We have

$$F(\rho) = \big(\rho(u_\rho) - \rho\big)D(u_\rho), \quad \rho(u_\rho) = \rho + \frac{F(\rho)}{D(u_\rho)},$$

by Lemma 1, and that $F(\rho^*) = 0$, we have

$$\delta_{\min}(\rho^* - \rho) \le F(\rho) - F(\rho^*) = F(\rho) \le \delta_{\max}(\rho^* - \rho),$$

since $\rho \le \rho^*$, $F(\rho) \ge \delta_{\min}(\rho^* - \rho) \ge 0$, thus $\rho(u_\rho) \ge \rho$, and by Lemma 2, $\rho \le \rho(u_\rho) \le \rho^*$. For the contraction, we have

$$\begin{aligned}
\rho^* - \rho(u_\rho) &= (\rho^* - \rho) - \frac{F(\rho)}{D(u_\rho)} \\
&\le (\rho^* - \rho) - \frac{F(\rho)}{\delta_{\max}} \\
&\le (\rho^* - \rho) - \frac{\delta_{\min}\,(\rho^* - \rho)}{\delta_{\max}} \\
&= \Big(1 - \frac{\delta_{\min}}{\delta_{\max}}\Big)(\rho^* - \rho) = \chi\,(\rho^* - \rho).
\end{aligned}$$

□

**Lemma 4.** *For all $\rho, \rho' \in \mathbb{R}$, every $x$, and every $a \neq u_{\rho'}(x)$,*

$$q_\rho\big(x, u_{\rho'}(x)\big) - q_\rho(x, a) \geq \Delta(\rho') - |\rho - \rho'|\, \delta_{\mathrm{spr}}.$$

*Consequently $\Delta(\rho) \geq \Delta(\rho') - |\rho - \rho'|\, \delta_{\mathrm{spr}}$ and hence, by symmetry, $|\Delta(\rho) - \Delta(\rho')| \leq |\rho - \rho'|\, \delta_{\mathrm{spr}}$: the greedy margin is $\delta_{\mathrm{spr}}$-Lipschitz in the rate. If moreover $|\rho - \rho'|\, \delta_{\mathrm{spr}} < \Delta(\rho')$, then $u_\rho = u_{\rho'}$ point-wise, under any tie-breaking. Setting $\rho' = \rho^*$ gives,*

$$\Delta(\rho) \geq \Delta^* - |\rho - \rho^*|\, \delta_{\mathrm{spr}}, \qquad \text{and} \qquad u_\rho = a^* \ \text{ whenever } \ |\rho - \rho^*|\, \delta_{\mathrm{spr}} < \Delta^*,$$

*in particular $|\rho - \rho^*| \leq \bar{\varepsilon}$ gives $u_\rho = a^*$ and $\Delta(\rho) \geq \Delta^*/2$.*

*Proof.* Compare the relative values at two reward-rates $\rho, \rho'$ similar to Lemma 1 gives

$$q_\rho(x, a) = q_{\rho'}(x, a) + (\rho' - \rho)\bar{\delta}(x, a), \quad \forall x \in \mathcal{X}, a \in \mathcal{A},$$

denote $u := u_{\rho'}(x)$ as the greedy policy, and let $a \neq u$, and substitute the equation above gives

$$q_\rho(x, u) - q_\rho(x, a) = \underbrace{\big[q_{\rho'}(x, u) - q_{\rho'}(x, a)\big]}_{(1)} + \underbrace{(\rho' - \rho)\big[\bar{\delta}(x, u) - \bar{\delta}(x, a)\big]}_{(2)}.$$

By definition (see Appendix B.1), $(1) \geq \Delta(\rho')$. For (2), both $\bar{\delta}$ terms lie within interval $[\min_b \bar{\delta}(x, b), \max_b \bar{\delta}(x, b)]$, whose length is at most $\delta_{\mathrm{spr}}$ (Appendix B.1), this gives

$$\big|(\rho' - \rho)\big[\bar{\delta}(x, u) - \bar{\delta}(x, a)\big]\big| \leq |\rho - \rho'|\, \big[\max_b \bar{\delta}(x, b) - \min_b \bar{\delta}(x, b)\big] \leq |\rho - \rho'|\, \delta_{\mathrm{spr}},$$

which implies

$$q_\rho(x, u) - q_\rho(x, a) \geq \Delta(\rho') - |\rho - \rho'|\, \delta_{\mathrm{spr}}.$$

If the right hand side is positive, then it implies that $u$ is an unique maximizer, thus $u_{\rho'}(x) = u_\rho(x)$, then taking the minimum over $x$ gives

$$\Delta(\rho) \geq \Delta(\rho') - |\rho - \rho'|\, \delta_{\mathrm{spr}},$$

If the right hand side is non-positive, then the above inequality still holds since $\Delta(\rho) \geq 0$ by construction (Appendix B.1). By symmetry, we also have that $\Delta(\rho') \geq \Delta(\rho) - |\rho - \rho'|\, \delta_{\mathrm{spr}}$, which implies

$$|\Delta(\rho) - \Delta(\rho')| \leq |\rho - \rho'|\, \delta_{\mathrm{spr}}.$$

This completes the proof for $\delta_{\mathrm{spr}}$-Lipschitz statement. If $|\rho - \rho'|\delta_{\mathrm{spr}} < \Delta(\rho')$, then $q_\rho(x, u) - q_\rho(x, a) > 0$, and since $u := u_{\rho'}(x)$, and by uniqueness of the greedy action assumption (Assumption 2), we have $u_{\rho'}(x) = u_\rho(x)$ for every $x$, i.e., the greedy policy for the two rates are the same.

For the last statement, substitute $\rho' = \rho^*$ gives the property

$$\Delta(\rho) \geq \Delta^* - |\rho - \rho^*|\, \delta_{\mathrm{spr}}, \qquad \text{and} \qquad u_\rho = a^* \ \text{ whenever } \ |\rho - \rho^*|\, \delta_{\mathrm{spr}} < \Delta^*,$$

and with $\bar{\varepsilon} = \Delta^*/(2\delta_{\mathrm{spr}})$, if $|\rho - \rho^*| \leq \bar{\varepsilon}$, then $|\rho - \rho^*|\delta_{\mathrm{spr}} \leq \Delta^*/2 < \Delta(\rho^*) = \Delta^*$, which implies $u_\rho = a^*$ and $\Delta(\rho) \geq \Delta^*/2$. □

Lemmas 1 to 3 are statements on the Dinkelbach's algorithm for fractional programming similar to those in Jagannathan (1966); Dinkelbach (1967); Schaible (1976).

### B.3 Main Results

**Theorem 3.** *Let $\gamma \in [0, 1]$ and $\rho_1 \in [0, \rho_{\max}]$. For every $k \geq 1$, let $u_{\rho_k}(x) \in \arg\max_a q_{\rho_k}(x, a)$ be a greedy policy under any tie-breaking and $\rho(u_{\rho_k})$ is the reward rate of the greedy policy, let $\tilde{\rho}_{k+1}$ be any value with the sequence satisfying the condition*

$$\frac{\rho(u_{\rho_k})}{1 + \gamma} \leq \tilde{\rho}_{k+1} \leq (1 + \gamma)\, \rho(u_{\rho_k}), \qquad \rho_{k+1} = \mathrm{clip}_{[0, \rho_{\max}]}\big(\tilde{\rho}_{k+1}\big), \tag{30}$$

*where $\mathrm{clip}_{[0, \rho_{\max}]}(z) := \min\{\max\{z, 0\}, \rho_{\max}\}$. Then, for every $k \geq 2$,*

$$\left(1 - \frac{R\gamma}{1 + \gamma}\right)\rho^* - \left(\frac{\chi}{1 + \gamma}\right)^{k-2} \rho_{\max} \leq \rho_k \leq (1 + \gamma)\rho^*, \tag{31}$$

*where $R = \delta_{\max}/\delta_{\min} \geq 1$. If $\gamma = 0$, then the sequence increases monotonically $\rho_2 \leq \rho_3 \leq \cdots \leq \rho^*$ and the iteration reaches $\rho_k = \rho^*$ after finite steps.*

*Proof.* Throughout the proof, we use the notation

$$N(\pi) := \mathbb{E}_p\Big[\sum_a \pi(a \mid x)\,\bar{r}(x,a)\Big], \qquad D(\pi) := \mathbb{E}_p\Big[\sum_a \pi(a \mid x)\,\bar{\delta}(x,a)\Big],$$

and $\rho(\pi) = N(\pi)/D(\pi)$ by definition, and

$$\begin{aligned}\mathbb{E}_p\Big[\sum_a \pi(a \mid x)\,q_\rho(x,a)\Big] &= N(\pi) \;-\; \rho\,D(\pi) \\ &= \rho(\pi)\,D(\pi) \;-\; \rho\,D(\pi) = \big(\rho(\pi)-\rho\big)\,D(\pi).\end{aligned}$$

By Lemma 1, we have

$$0 \le \rho(u_{\rho_k}) \le \rho^* \le \rho_{\max}.$$

We first note that the clip operation survives the upper and lower bounds. For upper bounds with a nonnegative right hand side: if $z \le B$ and $B \ge 0$, we have

$$\mathrm{clip}_{[0,\rho_{\max}]}(z) \le \max\{z, 0\} \le B,$$

and for lower bounds with a right-hand side at most $\rho_{\max}$: if $z \ge b$ with $b \le \rho_{\max}$, then

$$\mathrm{clip}_{[0,\rho_{\max}]}(z) \ge \min\{z,\, \rho_{\max}\} \ge \min\{b,\, \rho_{\max}\} = b.$$

Since the bounds survive, for notation simplicity, we use $\rho_{k+1}$ and $\tilde{\rho}_{k+1}$ interchangeably.

For $k \ge 1$, using the condition on the sequence in Equation (30), we have

$$\tilde{\rho}_{k+1} \le (1+\gamma)\,\rho(u_{\rho_k}) \;\le\; (1+\gamma)\,\rho^* \implies \rho_{k+1} \le (1+\gamma)\rho^*,$$

and this completes the upper bound in Equation (31). To prove the lower bound in Equation (31), we define

$$e_k := \rho^* - \rho_k, \quad B := \frac{R\gamma\rho^*}{1+\gamma}.$$

To complete the rest of the proof, we enumerate each of the two cases when $k \ge 2$: (i) *the below case* with $\rho_k \le \rho^*$. (ii) *the above case* with $\rho^* < \rho_k \le (1+\gamma)\rho^*$. For the below case with $\rho_k \le \rho^*$, using Lemma 3 gives

$$\rho(u_{\rho_k}) \ge \rho^* - \chi(\rho^* - \rho_k) = \rho^* - \chi e_k,$$

using the condition in Equation (30), we have $\rho_{k+1} \ge \rho(u_{\rho_k})/(1+\gamma)$, which gives

$$\rho^* - \rho_{k+1} \le \rho^* - \frac{\rho^* - \chi e_k}{1+\gamma} = \frac{(1+\gamma)\rho^* - \rho^* + \chi e_k}{1+\gamma} = \frac{\chi e_k + \gamma\rho^*}{1+\gamma},$$

thus we established the fact that $e_{k+1} \le \frac{\chi e_k + \gamma\rho^*}{1+\gamma}$ for the below case.

For the above case, $\rho^* < \rho_k \le (1+\gamma)\rho^*$ implies $0 < \rho_k - \rho^* \le \gamma\rho^*$. By Lemma 1 and the fact that $\rho^* < \rho_k$, we have

$$-\delta_{\max}(\rho_k - \rho^*) \le F(\rho_k) \le -\delta_{\min}(\rho_k - \rho^*) \le 0. \tag{32}$$

Combine together with the fact that $F(\rho_k)/D(u_{\rho_k}) \ge F(\rho_k)/\delta_{\min}$ gives

$$\frac{F(\rho_k)}{D(u_{\rho_k})} \ge -\frac{\delta_{\max}}{\delta_{\min}}(\rho_k - \rho^*), \tag{33}$$

which implies

$$\begin{aligned}\rho(u_{\rho_k}) &= \rho_k + \frac{F(\rho_k)}{D(u_{\rho_k})} \ge \rho_k - \frac{\delta_{\max}}{\delta_{\min}}(\rho_k - \rho^*) \\ &= \rho^* + (\rho_k - \rho^*) - \frac{\delta_{\max}}{\delta_{\min}}(\rho_k - \rho^*) \\ &= \rho^* - \left(\frac{\delta_{\max}}{\delta_{\min}} - 1\right)(\rho_k - \rho^*) \ge \rho^* - (R-1)\,\gamma\rho^*,\end{aligned}$$

where the first equality comes from the proof in Lemma 3. Use the equation above with the fact $\rho_{k+1} \geq \rho(u_{\rho_k})/(1+\gamma)$, we have

$$\begin{aligned}\rho^* - \rho_{k+1} &\leq \rho^* - \frac{\rho(u_{\rho_k})}{1+\gamma} \\ &\leq \rho^* - \frac{\rho^* - (R-1)\gamma\rho^*}{1+\gamma} \\ &= \rho^* \frac{(1+\gamma) - 1 + (R-1)\gamma}{1+\gamma} = \frac{R\gamma\rho^*}{1+\gamma} = B.\end{aligned} \tag{34}$$

Hence we established the fact that $e_{k+1} \leq B$ for the above case. The final proof is to show by induction that the following is true

$$e_k \leq \tilde{\chi}^{k-2}\rho_{\max} + B, \qquad k \geq 2, \tag{35}$$

where $\tilde{\chi} := \chi/(1+\gamma) \in [0, 1)$. For the base case $k = 2$, since $\rho_2 \geq 0$, we have $\rho^* - \rho_2 \leq \rho^* \leq \rho_{\max}$, which gives $e_2 \leq \rho_{\max} \leq \rho_{\max} + B$. For the induction step, we assume the claim is true for some $k \geq 2$ and enumerate each of the two cases $\rho_k \leq \rho^*$ and $\rho_k > \rho^*$. If $\rho_k > \rho^*$, then by Equation (34), we have $e_{k+1} \leq B \leq \tilde{\chi}^{k-1}\rho_{\max} + B$. If $\rho_k \leq \rho^*$, then

$$\begin{aligned}e_{k+1} &\leq \frac{\chi e_k + \gamma\rho^*}{1+\gamma} \leq \frac{\chi\left(\tilde{\chi}^{k-2}\rho_{\max} + B\right) + \gamma\rho^*}{1+\gamma} \\ &= \tilde{\chi}^{k-1}\rho_{\max} + \frac{\chi B + \gamma\rho^*}{1+\gamma},\end{aligned} \tag{36}$$

$$\frac{\chi B + \gamma\rho^*}{1+\gamma} = \frac{\gamma\rho^*}{(1+\gamma)^2}(\chi R + 1 + \gamma) \leq \frac{\gamma\rho^*}{(1+\gamma)^2}(R + R\gamma) = B, \tag{37}$$

where the last inequality comes from the identity that $\chi R + 1 = R$, so $\chi R + 1 + \gamma \leq R + R\gamma$. Combine Equation (36) with Equation (37), we have

$$e_{k+1} \leq \tilde{\chi}^{k-1}\rho_{\max} + \frac{\chi B + \gamma\rho^*}{1+\gamma} \leq \tilde{\chi}^{k-1}\rho_{\max} + B,$$

rearranging the terms gives the lower bound and completes the bound

$$\rho^*\left(1 - \frac{R\gamma}{1+\gamma}\right) - \left(\frac{\chi}{1+\gamma}\right)^{k-2}\rho_{\max} \leq \rho_k.$$

We now show that when $\gamma = 0$, then the iteration reaches $\rho^*$ in finite steps. If $\gamma = 0$, we have that every rate estimate equals to the current greedy policy's rate $\rho_{k+1} = \rho(u_{\rho_k})$, by the identity in Equation (33) and the upper bound $\rho_k \leq \rho^*$ in Equation (31), we have

$$\rho_{k+1} - \rho_k = \frac{F(\rho_k)}{D(u_{\rho_k})} \geq \frac{\delta_{\min}}{\delta_{\max}}(\rho^* - \rho_k) \geq 0,$$

hence the sequence increases monotonically. To prove that it reaches $\rho^*$ within finite steps, suppose for sake of contradiction that $\{\rho_k\}$ does not reach $\rho^*$ in finite steps, then we would have $\rho_k < \rho^*$ for every iteration $k = 2, \cdots, M+2$, which implies

$$\rho_{k+1} - \rho_k = \frac{F(\rho_k)}{D(u_{\rho_k})} \geq \frac{\delta_{\min}}{\delta_{\max}}(\rho^* - \rho_k) > 0,$$

which means the sequence $\{\rho_k\}$ is strictly monotonic. Since greedy policy is a deterministic policy, and there are $M = |\mathcal{A}|^{|\mathcal{X}|}$ possible deterministic policies, this means the possible number of reward rate is $|\{\rho(u) : u \in \mathcal{U}\}| \leq M$. Then we would have $\rho_2 < \rho_3 < \cdots < \rho_{M+2}$, which are $M+1$ distinct rates, a contradiction. □

**Proposition 1.** *Let $\bar{\rho} \in \mathbb{R}$ be any rate with $\Delta(\bar{\rho}) > 0$, write $u := u_{\bar{\rho}}$ for the greedy policy at $\bar{\rho}$, which is then unique at every context, and run the algorithm from $\theta^{(0)}$ with $\pi_0 := \pi_{\theta^{(0)}}(u(x) \mid x)$. For every $x$ and $k \geq 0$,*

$$1 - \pi_{\theta^{(k)}}(u(x) \mid x) \;\leq\; \frac{1-\pi_0}{\pi_0}\,\exp\left(-\eta\,\Delta(\bar{\rho})\,k\right), \tag{38}$$

*in particular $1 - \pi_{\theta^{(k)}}(u(x) \mid x) \leq (|\mathcal{A}| - 1)\exp\left(-\eta\,\Delta(\bar{\rho})\,k\right)$ when $\pi_0 = 1/|\mathcal{A}|$ (policy is initially uniform).*

*Proof.* We use same abbreviations as before, where the greedy policy is denoted as $u = u_{\hat{\rho}}(x)$. By definition of the margin, we have

$$q_{\bar{\rho}}(x,u) - q_{\bar{\rho}}(x,a) \geq q_{\bar{\rho}}(x,u) - \max_{b \neq u} q_{\bar{\rho}}(x,b) \geq \Delta(\bar{\rho}) > 0, \qquad \text{for every } a \neq u,$$

the algorithm updates the logit for each step by

$$\theta_x^{(n+1)} = \theta_x^{(n)} + \eta\Big(q_{\bar{\rho}}(x,\cdot) - \big(\boldsymbol{\pi}^{(n)\top} q_{\bar{\rho}}(x,\cdot)\big)\,\mathbf{1}\Big),$$

and the logit between actions $u$ and $a$ is

$$\begin{aligned}\theta_{x,u}^{(n+1)} - \theta_{x,a}^{(n+1)} &= \Big(\theta_{x,u}^{(n)} + \eta\, q_{\bar{\rho}}(x,u) - \eta\, \boldsymbol{\pi}^{(n)\top} q_{\bar{\rho}}(x,\cdot)\Big) \\ &\qquad - \Big(\theta_{x,a}^{(n)} + \eta\, q_{\bar{\rho}}(x,a) - \eta\, \boldsymbol{\pi}^{(n)\top} q_{\bar{\rho}}(x,\cdot)\Big) \\ &= \big(\theta_{x,u}^{(n)} - \theta_{x,a}^{(n)}\big) \;+\; \eta\big(q_{\bar{\rho}}(x,u) - q_{\bar{\rho}}(x,a)\big),\end{aligned}$$

write $d^{(n)} := \theta_{x,u}^{(n)} - \theta_{x,a}^{(n)}$ for the logit gap and $c \;:=\; \eta\big(q_{\bar{\rho}}(x,u) - q_{\bar{\rho}}(x,a)\big)$ for the update gap, the parameter update rule thus becomes a recursion of the form $d^{(n+1)} = d^{(n)} + c$. Writing as a telescoping sum over $n = 0, \ldots, k-1$ gives

$$d^{(k)} - d^{(0)} = \sum_{n=0}^{k-1} \big(d^{(n+1)} - d^{(n)}\big) = \sum_{n=0}^{k-1} c \;=\; k\,c,$$

thus we have

$$\theta_{x,u}^{(k)} - \theta_{x,a}^{(k)} = \theta_{x,u}^{(0)} - \theta_{x,a}^{(0)} + \eta\, k\big(q_{\bar{\rho}}(x,u) - q_{\bar{\rho}}(x,a)\big),$$

since our distribution is parameterized through softmax, we have

$$\frac{\pi_\theta(a \mid x)}{\pi_\theta(u \mid x)} = \frac{\exp(\theta_{x,a}) / \sum_b \exp(\theta_{x,b})}{\exp(\theta_{x,u}) / \sum_b \exp(\theta_{x,b})} = \exp(\theta_{x,a} - \theta_{x,u}),$$

fix $a \neq u$, and apply the two equations together, we have

$$\begin{aligned}\frac{\pi_{\theta^{(k)}}(a \mid x)}{\pi_{\theta^{(k)}}(u \mid x)} &= \exp\big(\theta_{x,a}^{(k)} - \theta_{x,u}^{(k)}\big) \\ &= \exp\Big(\theta_{x,a}^{(0)} - \theta_{x,u}^{(0)} - \eta\, k\big(q_{\bar{\rho}}(x,u) - q_{\bar{\rho}}(x,a)\big)\Big) \\ &= \frac{\pi_{\theta^{(0)}}(a \mid x)}{\pi_{\theta^{(0)}}(u \mid x)}\; \exp\Big(-\eta\, k\big(q_{\bar{\rho}}(x,u) - q_{\bar{\rho}}(x,a)\big)\Big),\end{aligned}$$

observe that

$$-\eta\, k\big(q_{\bar{\rho}}(x,u) - q_{\bar{\rho}}(x,a)\big) \;\leq\; -\eta\, k\, \Delta(\bar{\rho}),$$

thus

$$\frac{\pi_{\theta^{(k)}}(a \mid x)}{\pi_{\theta^{(k)}}(u \mid x)} \;\leq\; \frac{\pi_{\theta^{(0)}}(a \mid x)}{\pi_{\theta^{(0)}}(u \mid x)}\; \exp(-\eta\Delta(\bar{\rho})k),$$

taking the sum over all $a \neq u$ gives the following for the left hand side

$$\sum_{a \neq u} \frac{\pi_{\theta^{(k)}}(a \mid x)}{\pi_{\theta^{(k)}}(u \mid x)} = \frac{\sum_{a \neq u} \pi_{\theta^{(k)}}(a \mid x)}{\pi_{\theta^{(k)}}(u \mid x)} = \frac{1 - \pi_{\theta^{(k)}}(u \mid x)}{\pi_{\theta^{(k)}}(u \mid x)},$$

and the following for the right hand side

$$\sum_{a \neq u} \frac{\pi_{\theta^{(0)}}(a \mid x)}{\pi_{\theta^{(0)}}(u \mid x)}\; \exp(-\eta\Delta(\bar{\rho})k) = \frac{1 - \pi_0}{\pi_0}\; \exp(-\eta\Delta(\bar{\rho})k),$$

together we have

$$\frac{1 - \pi_{\theta^{(k)}}(u \mid x)}{\pi_{\theta^{(k)}}(u \mid x)} \;\leq\; \frac{1 - \pi_0}{\pi_0}\; \exp(-\eta\Delta(\bar{\rho})k),$$

multiplying both sides by $\pi_{\theta^{(k)}}(u \mid x)$ gives

$$\begin{aligned} 1 - \pi_{\theta^{(k)}}(u \mid x) &= \pi_{\theta^{(k)}}(u \mid x)\, \frac{1 - \pi_{\theta^{(k)}}(u \mid x)}{\pi_{\theta^{(k)}}(u \mid x)} \\ &\le \pi_{\theta^{(k)}}(u \mid x)\, \frac{1 - \pi_0}{\pi_0}\, \exp(-\eta \Delta(\bar{\rho}) k) \\ &\le \frac{1 - \pi_0}{\pi_0}\, \exp(-\eta \Delta(\bar{\rho}) k), \end{aligned}$$

and hence completes the proof. □

**Corollary 1.** *Let $\bar{\rho} \in \mathbb{R}$ be any rate with smallness condition $|\bar{\rho} - \rho^*| \le \bar{\varepsilon}$, $\bar{\varepsilon} = \Delta^*/(2\delta_{\mathrm{spr}})$. Then $u_{\bar{\rho}} = a^*$, for every $x$, $a \neq a^*(x)$, $k \ge 0$*

$$1 - \pi_{\theta^{(k)}}(a^*(x) \mid x) \le \frac{1 - \pi_0}{\pi_0}\, \exp\big(-\eta\, \Delta^* k/2\big), \tag{39}$$

*in particular $1 - \pi_{\theta^{(k)}}(a^*(x) \mid x) \le (|\mathcal{A}| - 1) \exp(-\eta \Delta^* k/2)$ at the zero initialization.*

*Proof.* Apply Lemma 4 with Proposition 1. □

**Theorem 4.** *Let $\rho_1, \rho_2, \ldots$ be a sequence of reward rates. Denote $\theta^{(k)}$ as the parameter for the softmax logit after $k$ steps from the zero initialization and $\bar{\rho}_k = \sum_{n \le k} \rho_n / k$, then for every $x$, $k \ge 1$, the policy $\pi^{(k)}$ is a Boltzmann policy of $q_{\bar{\rho}_k}(x, a)$ with inverse temperature $\eta k$, and if $\rho_{k+1} = \rho(u_{\bar{\rho}_k})$, $\rho_1 \in [0, \rho^*]$, then*

$$\rho^* - \bar{\rho}_k \;\le\; (\rho^* - \bar{\rho}_1)\, \exp(1 - \chi)\, (k+1)^{-(1-\chi)}, \tag{40}$$

*thus $\bar{\rho}_k$ converges to $\rho^*$ at $O(k^{-(1-\chi)})$. Moreover, for every*

$$k \;\ge\; k_0 \;:=\; \left\lceil \big( \exp(1 - \chi)\, (\rho^* - \bar{\rho}_1)/\bar{\varepsilon} \big)^{1/(1-\chi)} \right\rceil, \tag{41}$$

*the greedy policy at the running average is optimal, $u_{\bar{\rho}_k} = a^*$ point-wise, and for every $x$*

$$1 - \pi^{(k)}(a^*(x) \mid x) \;\le\; (|\mathcal{A}| - 1)\, \exp\big(-\eta\, \Delta^* k/2\big). \tag{42}$$

*Proof.* Observe that the relative value $q_{\rho_n}(x, a)$ for action $a$ and context $x$ at step $n$ can be expressed as

$$\begin{aligned} \sum_{n \le k} q_{\rho_n}(x, a) = \sum_{n \le k} \big( \bar{r}(x, a) - \rho_n\, \bar{\delta}(x, a) \big) \;&=\; k\, \bar{r}(x, a) \;-\; \Big( \sum_{n \le k} \rho_n \Big) \bar{\delta}(x, a) \\ &= k \big( \bar{r}(x, a) - \bar{\rho}_k\, \bar{\delta}(x, a) \big) = k\, q_{\bar{\rho}_k}(x, a), \end{aligned}$$

By the parameter update rule for NPG with a softmax policy $\theta'_x = \theta_x + \eta(q - \langle \boldsymbol{\pi}, q \rangle \mathbf{1})$, fix the context, and expand the terms through the recursion gives

$$\begin{aligned} \theta_x^{(k)} &= \eta \sum_{n \le k} q_{\rho_n}(x, \cdot) \;-\; \eta \left( \sum_{n \le k} s_n \right) \mathbf{1} \qquad s_n := \big\langle \boldsymbol{\pi}^{(n-1)}, q_{\rho_n}(x, \cdot) \big\rangle \\ &= \eta\, k\, q_{\bar{\rho}_k}(x, \cdot) \;+\; c_k(x)\, \mathbf{1} \qquad c_k(x) := -\eta \sum_{n \le k} s_n, \end{aligned}$$

when the logit $\theta_x^{(k)}$ is feed into the softmax function, since softmax is invariant to addition of constants, the $c_k(x)$ term is ignored, thus the policy becomes the Boltzmann policy of $\eta\, k\, q_{\bar{\rho}_k}(x, a)$ and completes the first part of the proof. For the second part, first define the estimated rate as $k \bar{\rho}_k = \sum_{n \le k} \rho_n$, then

$$\bar{\rho}_{k+1} = \frac{1}{k+1} \sum_{n \le k+1} \rho_n = \frac{k\, \bar{\rho}_k + \rho_{k+1}}{k+1} = \frac{k}{k+1}\, \bar{\rho}_k \;+\; \frac{1}{k+1}\, \rho_{k+1},$$

which is

$$\bar{\rho}_{k+1} = (1-\alpha_k)\bar{\rho}_k + \alpha_k\,\rho_{k+1}, \qquad \alpha_k = 1/(k+1). \tag{43}$$

Since the algorithm only accepts samples selected by $\arg\max_a \theta_{x,a}$, we assume that the rate estimate equals to the greedy policy's rate under $\bar{\rho}_k$, i.e., $\rho_{k+1} = \rho(u_{\bar{\rho}_k})$. Since $\bar{\rho}_k \le \rho^*$ at $k = 1$ by assumption, Lemma 3 gives that $\rho(u_{\bar{\rho}_k}) \ge \bar{\rho}_k$ and $\rho^* - \rho(u_{\bar{\rho}_k}) \le \chi(\rho^* - \bar{\rho}_k)$. $\bar{\rho}_{k+1}$ is a convex combination of $\bar{\rho}_k$ and $\rho(u_{\bar{\rho}_k})$ by Equation (43) and we have $\bar{\rho}_k \le \bar{\rho}_{k+1} \le \rho^*$. Subtracting Equation (43) from $\rho^*$ gives

$$\begin{aligned}
\rho^* - \bar{\rho}_{k+1} &= \rho^* + \alpha_k\rho^* - \alpha_k\rho^* - (1-\alpha_k)\bar{\rho}_k - \alpha_k\rho_{k+1} \\
&= (1-\alpha_k)(\rho^* - \bar{\rho}_k) + \alpha_k(\rho^* - \rho_{k+1}) \\
&\le (1-\alpha_k)(\rho^* - \bar{\rho}_k) + \alpha_k\chi(\rho^* - \bar{\rho}_k) = \big(1 - \alpha_k(1-\chi)\big)(\rho^* - \bar{\rho}_k).
\end{aligned}$$

Denote $e_k = \rho^* - \bar{\rho}_k$, $c := 1 - \chi = \delta_{\min}/\delta_{\max} \in (0, 1]$, and expand the recursion gives

$$\begin{aligned}
e_k &\le e_1 \prod_{m=2}^{k} \big(1 - c/m\big) \le e_1 \exp\left(-c \sum_{m=2}^{k} 1/m\right) \\
&\le e_1 \exp\big(c - c\log(k+1)\big) = e_1 \exp(c)\,(k+1)^{-c},
\end{aligned}$$

where we used the fact $1 - z \le \exp(-z)$ and $\sum_{m \le k} 1/m \ge \log(k+1)$.

Since $\rho_{k+1} = \rho(u_{\bar{\rho}_k}) \le \rho^*$, $\bar{\rho}_{k+1} \le (1-\alpha_k)\bar{\rho}_k + \alpha_k\rho^* < \rho^*$ whenever $\bar{\rho}_k < \rho^*$, so by induction $\rho_1 < \rho^*$ keeps the strict inequality $\bar{\rho}_k < \rho^*$ to every $k > 1$.

For the last part, the second part gives $\bar{\rho}_k \le \bar{\rho}_{k+1} \le \rho^*$, so the error $\rho^* - \bar{\rho}_k$ is non-increasing in $k$ and it suffices to find the first index at which it falls below $\bar{\varepsilon}$. By Equation (40), $\rho^* - \bar{\rho}_k \le \bar{\varepsilon}$ holds when

$$(k+1)^{1-\chi} \ge \exp(1-\chi)\,(\rho^* - \bar{\rho}_1)/\bar{\varepsilon},$$

thus we need

$$k \ge \big(\exp(1-\chi)\,(\rho^* - \bar{\rho}_1)/\bar{\varepsilon}\big)^{1/(1-\chi)} - 1,$$

thus we can set $k_0$ to be the constant stated in Equation (41), and it then holds at every subsequent step $k \ge k_0$. When $k \ge k_0$, since $|\bar{\rho}_k - \rho^*| \le \bar{\varepsilon}$, we can apply Lemma 4, which gives $u_{\bar{\rho}_k} = u_{\rho^*} = a^*$ and $\Delta(\bar{\rho}_k) \ge \Delta^*/2$. Similar to the steps in the proofs for Proposition 1 and Corollary 1, the probability ratio is

$$\begin{aligned}
\frac{\pi^{(k)}(a \mid x)}{\pi^{(k)}(a^*(x) \mid x)} &= \exp\Big(-\eta\,k\big(q_{\bar{\rho}_k}(x, a^*(x)) - q_{\bar{\rho}_k}(x, a)\big)\Big) \\
&\le \exp\big(-\eta\,k\,\Delta(\bar{\rho}_k)\big) \\
&\le \exp\big(-\eta\,\Delta^*\,k/2\big),
\end{aligned}$$

sum over all actions $a \ne a^*(x)$ gives

$$\frac{1 - \pi^{(k)}(a^*(x) \mid x)}{\pi^{(k)}(a^*(x) \mid x)} \le (|\mathcal{A}| - 1)\exp\big(-\eta\,\Delta^*\,k/2\big),$$

which gives Equation (42) after multiplying both sides by $\pi^{(k)}(a^*(x) \mid x) \le 1$ and completes the proof. □

# C IMPLEMENTATION

## C.1 BANDIT EXPERIMENTS

This section gives the full specification of the bandit experiments in Section 3.3.

### C.1.1 BANDIT BASELINES

**C-UCB with theoretical constants.** The algorithm of György et al. (2007), implemented faithfully for all the constants.

**C-UCB with tuned constants.** Identical in every respect, except that the two scale constants of Equations (27) and (28) are replaced by free parameters $c_1$ and $c_2$ selected via grid search.

**C-UCB with approximation.** C-UCB involves enumeration over all policies for reward-rate estimation. This is infeasible when we have many arms and contexts. As an approximation, we draw 2048 policies uniformly random from policies constructed from visited arms and contexts, and enumerate on these 2048 policies.

**NPG.** The reward-rate learner keeps one independent softmax head per context: a tabular parameter matrix $\theta \in \mathbb{R}^{|\mathcal{X}|\times(K+1)}$ with one free logit per $(x, a)$ pair, initialized to zero so that $\pi_\theta(a \mid x) = 1/(K+1)$ is uniform, and

$$\pi_\theta(a \mid x) \;=\; \frac{\exp(\theta_{x,a})}{\sum_b \exp(\theta_{x,b})}. \tag{44}$$

At each round the learner observes $x$, samples $a \sim \pi_\theta(\cdot \mid x)$, and receives $(r, \delta)$, and its regret is incurred by its own action choices. The update rule is the natural gradient of the relative reward

$$\theta_x \leftarrow \theta_x + \eta\, \boldsymbol{z}, \quad \nabla_{\theta_x} J(\theta) = \mathbf{F}(\theta_x)\boldsymbol{z}, \tag{45}$$

where the Fisher information matrix $\mathbf{F}$ is

$$\mathbf{F}(\theta_x) = \mathrm{diag}(\boldsymbol{\pi}) - \boldsymbol{\pi}\boldsymbol{\pi}^\top, \tag{46}$$

which simplifies to adding the advantage to the logits directly. Writing $\boldsymbol{\pi} = \pi_\theta(\cdot \mid x)$, $\boldsymbol{s}$ for the vector with entries $s_a = \mathbb{E}[\, r - \rho^*\delta \mid x, a\,]$, $J = \boldsymbol{\pi}^\top \boldsymbol{s}$ and $\boldsymbol{a} = \boldsymbol{s} - \big(\boldsymbol{\pi}^\top \boldsymbol{s}\big)\mathbf{1}$ be the advantages

$$\frac{\partial \pi_a}{\partial \theta_c} = \pi_a\big(\mathbb{1}\{a = c\} - \pi_c\big) \quad \Longrightarrow \quad \nabla_{\theta_x} J = \boldsymbol{\pi} \odot \boldsymbol{a}, \tag{47}$$

solving for $\mathbf{F}\boldsymbol{z} = \boldsymbol{\pi} \odot \boldsymbol{a}$ gives $\boldsymbol{z} = \boldsymbol{a}$ as one of the solutions, thus the update rule is

$$\theta_x \leftarrow \theta_x + \eta\, A_\theta(x, \cdot).$$

The advantage is needed for every arm while a bandit learner observes only the arm it played, so we keep a per-cell buffer of the raw sums $S^r_{x,a}$ and $S^\delta_{x,a}$ with counts $n_{x,a}$, and use them to estimate the expected relative reward under current estimated rate $\hat{\rho}_t$,

$$\widehat{Q}_{x,a} \;=\; \begin{cases} \big(S^r_{x,a} - \hat{\rho}_t\, S^\delta_{x,a}\big)/n_{x,a}, & n_{x,a} > 0, \\ 0, & n_{x,a} = 0, \end{cases} \qquad \theta_x \;\leftarrow\; \theta_x + \eta\big(\widehat{Q}_x - (\boldsymbol{\pi}^\top \widehat{Q}_x)\mathbf{1}\big). \tag{48}$$

The rate $\hat{\rho}_t$ is supplied by the online NIW estimator.

**Vanilla policy gradient.** Everything is kept same as NPG, except the gradient update is

$$\theta_x \leftarrow \theta_x + \eta \nabla_{\theta_x} J(\theta),$$

which gives the update rule

$$\theta_x \leftarrow \theta_x + \eta\, \boldsymbol{\pi} \odot \big(\hat{Q}_x - (\boldsymbol{\pi}^\top \hat{Q}_x)\mathbf{1}\big).$$

**The NIW rate estimator.** The estimate $\hat{\rho}_t$ is fitted online on pairs of accumulated reward and accumulated time, mirroring the way the rate is formed from per-slot accumulators in MLE-Bench and NanoGPT (Section 4.1). It is refitted every 8 steps, and $\hat{\rho}_t$ is the 95th percentile among posterior-predictive samples (see Appendix C.2). The accumulators are *gated*, in that a sample enters an accumulator only when the arm played was the current $\arg\max_a \theta_{a,x}$, so that the estimate describes the greedy policy. They are also *age-bounded*, in that a slot restarts once it is older than $A$ steps, with $A$ tuned with hyperparameter search. We note that this estimator does not guarantee exact estimation of the greedy policy's reward rate. Rather, it serves as a test case to see how our algorithm behaves when the greedy reward rate has approximation error, which is the case in empirical settings.

### C.1.2 Experiment Setup

**Setup.** At each decision epoch the learner selects an action $a$ and observes a reward $r \geq 0$ together with a time $\delta > 0$. The pair is drawn jointly, so the reward and the time it costs may be dependent. Performance is measured against the optimal reward rate $\rho^*$, and we report the realized regret of Equation (6) at step $t$. Each context and action pair consists of a mean reward and time parameter, forming a table of size $(K, |\mathcal{X}|)$. The setup for these mean parameters are described below. To ensure we do not introduce bias in terms of ordering of the arms and contexts, we perform a permutation on $K$ arm indices and $|\mathcal{X}|$ context indices for every trial.

**Arm design.** The $K + 1$ actions available in a context are the SKIP action and two structurally different kinds of arm, which are specified in opposite directions. The four designed arms fix the pair $(\mathbb{E}[r(a)], \mathbb{E}[\delta(a)])$ and let the rate follow:

- BEST arm $(0.50, 1.0)$ at rate $0.500$.
- DECOY $(0.57, 1.2)$ at rate $0.475$, carrying higher raw reward at a $5\%$ relative rate deficit.
- TRAP $(0.90, 3.0)$ at rate $0.300$, the highest raw reward in the context at the worst rate.
- FAST arm $(0.21, 0.6)$ at rate $0.350$ that is cheap but weak.

The DECOY and the TRAP both exist to punish an agent that maximizes reward rather than reward per unit of time. The remaining $K - 4$ filler arms fix the rate and derive the reward and time from it

$$\begin{gathered} \zeta_j = \beta_x\Big(0.42 - \tfrac{j}{K-5}\big(0.42 - 0.18\big)\Big), \quad \mathbb{E}[\delta(4+j)] = d_j, \\ d_j \sim \mathrm{Unif}(0.70,\ 2.00), \quad \mathbb{E}[r(4+j)] = \zeta_j\, d_j, \end{gathered} \tag{49}$$

for $j = 0, \ldots, K - 5$, where $\beta_x$ is the reward scale of the context (see below) and each $d_j$ is drawn independently. This means the reward rate is monotone but reward and time are unordered, so neither quantity on its own identifies a good arm. No filler arm is ever optimal, because the sequence's maximum is at $0.42\,\beta_x$, strictly below the BEST arm at $0.50\,\beta_x$. Filler arms therefore widen the action set at controlled rate spacing rather than competing for the optimum, which makes $K$ an exploration-difficulty axis.

**Context Design.** A context $x \in \mathcal{X}$ is drawn uniformly at each epoch and adds a SKIP action with $(0, 0.35)$. A context is summarized by the reward scale $\beta_x$, which multiplies every reward and leaves time untouched; since the rate is reward divided by time, $\beta_x$ scales the rate as well. Four contexts are designed:

- MARGINAL ($\beta_x = 1.00$) reproduces the arm design above and serves as the baseline.
- RICH ($\beta_x = 1.50$) holds the highest multiplier.
- POOR ($\beta_x = 0.32$) so that SKIP can be considered.
- COUPLING is not a rescaling at all. Instead, it replaces the four designed arms with $(1.80, 2.00)$, $(1.00, 1.00)$, $(1.60, 3.0)$ and $(0.20, 0.6)$, a slow-but-rich arm at rate $0.90$ against a fast-but-lean arm at rate $1.00$.

Maximizing the rate within COUPLING takes the fast arm while the optimal policy takes the slow one, which is why the problem does not decompose into $|\mathcal{X}|$ independent bandits. This property holds as long as $\rho^* < 0.8$, which is the case for all of our simulations. Any context beyond these four is filler, where $\beta_x$'s are uniformly binned midpoints between $[0.42, 1.35]$, the range between POOR and RICH.

**Environment families.** Four families vary the dependence between reward and time while holding the marginal means fixed. Let $\mu_a^r = \mathbb{E}[r_a]$ and $\mu_a^\delta = \mathbb{E}[\delta_a]$ be the designed means of arm $a$. We fix $\sigma_r = 0.20$, a time noise fraction $\varsigma = 0.15$, a log-scale spread $\sigma_{\log} = 0.5$, and a time floor $\delta_{\min} = 0.5$. Every family draws a pair of standard normals $z_1, \xi \sim \mathcal{N}(0,1)$ and couples them through a Gaussian copula

$$z_2 \;=\; \varrho\, z_1 \;+\; \sqrt{1-\varrho^2}\,\xi, \qquad \operatorname{corr}(z_1, z_2) = \varrho, \tag{50}$$

so that $\varrho$ alone controls the reward–time dependence while both marginals are unchanged.

**E1 (independent), $\varrho = 0$.** Reward and time are rectified normals drawn independently,

$$r_a = \max\big(\mu_a^r + \sigma_r z_1, 0\big), \qquad \delta_a = \max\big(\mu_a^\delta + \varsigma\,\mu_a^\delta z_2,\ \delta_{\min}\big). \tag{51}$$

**E2 (positively coupled), $\varrho = +0.8$** and **E3 (negatively coupled), $\varrho = -0.8$.** Identical marginals to Equation (51), with $z_2$ generated by Equation (50). Under E2 a high-reward draw tends to also be slow, so the empirical rate $r/\delta$ is compressed and a learner that ranks arms by raw reward is punished less; under E3 high reward tends to arrive fast, which inflates the apparent spread in rates.

**E4 (lognormal), $\varrho = +0.5$ on the log scale.** Both coordinates are lognormal with heavy right tails. Note that we take the log on the mean parameter so that the expectation $\mathbb{E}[r_a], \mathbb{E}[\delta_a]$ still equals to $\mu_a^r, \mu_a^\delta$:

$$r_a = \exp\Big(\log \mu_a^r - \tfrac{\sigma_{\log}^2}{2} + \sigma_{\log} z_1\Big), \qquad \delta_a = \max\Big(\exp\big(\log \mu_a^\delta - \tfrac{\sigma_{\log}^2}{2} + \sigma_{\log} z_2\big),\ \delta_{\min}\Big). \tag{52}$$

The rectification and the floor $\delta_{\min}$ make the realized marginals differ slightly from $(\mu_a^r, \mu_a^\delta)$, so we do not rely on the design values when computing $\rho^*$ (See Equation (58)). Every reported mean is instead computed in closed form, using

$$\mathbb{E}[\max(X, c)] = c\,\Phi(a) + m\big(1 - \Phi(a)\big) + s\,\phi(a), \quad a = \tfrac{c-m}{s}, \tag{53}$$

$$\mathbb{E}[\max(Y, c)] = c\,\Phi(b) + m\big(1 - \Phi(b - \sigma_{\log})\big), \quad b = \tfrac{\log c - \mu}{\sigma_{\log}}, \tag{54}$$

for a rectified normal $X \sim \mathcal{N}(m, s^2)$ and a floored lognormal $Y$ with log-scale $\sigma_{\log}$. $\phi, \Phi$ are PDF and CDF of Gaussian, respectively. The means are therefore deterministic and do not require Monte Carlo approximation, and $\rho^*$ follows from Equation (58).

**Ground truth.** The general average-reward SMDP optimality equation reads

$$V^*(x) = \max_{a \in \mathcal{A}} \Big\{ \mathbb{E}[r(x,a)] - \rho^*\, \mathbb{E}[\delta(x,a)] + \sum_{x' \in \mathcal{X}} p(x' \mid x, a)\, V^*(x') \Big\}. \tag{55}$$

In the bandit case the next context is drawn independently of the current context and of the action taken, so $p(x' \mid x, a) = p(x')$ and the continuation term collapses to a constant $c := \sum_{x'} p(x') V^*(x')$ that depends on neither $x$ nor $a$. Writing

$$q^*(x,a) := \mathbb{E}[r(x,a)] - \rho^*\, \mathbb{E}[\delta(x,a)], \tag{56}$$

Equation (55) becomes

$$V^*(x) = \max_{a \in \mathcal{A}} q^*(x,a) + c. \tag{57}$$

Averaging against $p(x)$ on both sides and cancelling $c$ gives the condition that $\rho^*$ is the unique root of

$$F(\rho) \;=\; \sum_x p(x) \max_{a \in \mathcal{A}} \big(\mathbb{E}[r(x,a)] - \rho\, \mathbb{E}[\delta(x,a)]\big) \;=\; 0. \tag{58}$$

$F$ is a maximum of affine functions of $\rho$, hence convex, piecewise linear and strictly decreasing, so the root is unique and we find $\rho^*$ by bisection. Empirically, the residual is small $|F(\rho^*)| \sim 10^{-17}$.

**Evaluation.** Every instance is run to horizon $T = 30{,}000$ over all four families, in two studies whose arm and context counts differ:

- The small-scale study at $|\mathcal{X}| = 4$ with $K \in \{5, 10, 20, 30\}$.
- The larger-scale study at $|\mathcal{X}| \in \{8, 16, 32, 64\}$ with $K \in \{70, 100, 130, 160\}$.

Hyperparameters are selected by grid search on tuning seeds disjoint from the evaluation seeds, taking the *median* final regret over the tuning seeds, and the selected configuration is then run on 100 held-out evaluation seeds. Every step that took place counts as part of the regret calculation, including the burn-in phase (for instance, to have a nonzero value of counts for C-UCB). Confidence bands are pointwise percentile bootstraps of the mean over trials.

### C.2 Reward-rate Predictor

We introduce the reward-rate estimator we use for MLE. For $f_\rho$, we use a NIW prior with joint Gaussian likelihood (Murphy, 2012) fitted from historical samples observed in buffer $\mathcal{B}$. We model the pair as $(r_{\mathrm{acc}}, \log \delta_{\mathrm{acc}})$, where $r_{\mathrm{acc}}$ and $\delta_{\mathrm{acc}}$ are the accumulated reward and time, and we take the logarithm because we need time to be positive. Writing $z_i = (r_{\mathrm{acc},i},\, \log \delta_{\mathrm{acc},i})$ for the $i^{\text{th}}$ of the $n$ samples available at step $t$, where $n$ is the batch size. The likelihood and its conjugate prior are

$$z_i \mid \mu, \Sigma \sim \mathcal{N}(\mu, \Sigma), \qquad (\mu, \Sigma) \sim \mathrm{NIW}\big(\mu_{t-1},\, \lambda\kappa_{t-1},\, \lambda\nu_{t-1},\, \lambda\Psi_{t-1}\big), \tag{59}$$

where $\mu_{t-1}$ is the prior mean of $\mu$, $\Psi_{t-1}$ is the prior scatter matrix of $\Sigma$, and $\kappa_{t-1}$ and $\nu_{t-1}$ are pseudo-counts. The prior at step $t$ is the previous step's posterior discounted by a forgetting factor $\lambda \in (0, 1]$, which shrinks those pseudo-counts so that old evidence decays geometrically, and the posterior predictive is a Student-$t$ with $\nu_t - 1$ degrees of freedom, where $\nu_t = \lambda\nu_{t-1} + n$ is the posterior count after this step's $n$ samples are absorbed. Reward rate is estimated by sampling $M$ batches of $n$ draws each from this predictive. Writing the $m^{\text{th}}$ batch as $\{(x_{m,j},\, y_{m,j})\}_{j=1}^{n}$, each batch yields one rate, and $\hat{\rho}_{t+1}$ is the 95th percentile of them,

$$\rho_m = \frac{\sum_{j=1}^{n} x_{m,j}}{\sum_{j=1}^{n} \exp(y_{m,j})}, \qquad \hat{\rho}_{t+1} = \mathrm{Quantile}_{0.95}\big(\{\rho_m\}_{m=1}^{M}\big). \tag{60}$$

### C.3 MLE-Bench

#### C.3.1 Environment, Reward and Self-improvement

**Execution.** Every action $a$ is a code block generated by the policy (Qwen3.5-4B). The code block is executed in its own sandbox subprocess environment with its own working directory, under a memory budget of 16GB and a compute budget of 2 CPUs. The subprocess is killed when it reaches timeout $\delta_{\max}$. We set $\delta_{\max} = 600$ seconds, except for nomad2018, pizza, tabular-2021 and tabular-2022, where the dataset is large enough that 600 seconds admits almost no honest attempt, so we use $\delta_{\max} = 1200$ seconds for these tasks. Time $\delta$ is floored at `clip_time` $= 0.1$ seconds before it is used anywhere, so that no sample can claim an unboundedly large rate. We train with a batch size of 128. Thus, assuming full parallelization at each step, we require 16×128 = 2048GB of memory with 256 CPUs. This amount of resource is large in academic setting, so we use a first-in-first-out queue that processes the 128 samples by 16 samples running parallel at a time through Ray.

**Reward.** A run is a *valid submission* when the script terminates before $\delta_{\max}$, generates a submission file, and that file is accepted by the MLE-Bench grader. The grader returns the competition's own metric. For the 15 tasks whose metric is higher-is-better and bounded, we use performance metric (such as AUC) directly as the reward. For the remaining 7 tasks, where the metric is lower-is-better and has no bounded value (such as log loss), we use the position of the submission on the MLE-Bench leaderboard normalized as percentiles as reward. When there is a tie on the leaderboard, we use the mid-point position.

$$r_{t+1} \;=\; \frac{1}{N}\sum_{i=1}^{N}\Big(\mathbb{1}\{\ell_i \prec g\} \;+\; \tfrac{1}{2}\,\mathbb{1}\{\ell_i = g\}\Big) \;\in\; [0, 1], \tag{61}$$

where $\ell_1, \ldots, \ell_N$ are the leaderboard entries and $\prec$ orders them in the competition's own direction. This restores the higher-is-better direction and bounds the reward in $[0, 1]$. The 7 tasks using leaderboard score are exactly the seven marked with $(\downarrow)$ in Table 1.

**Failures.** If a script fails to obtain a valid submission, we define it as a failure. We use a penalty of $-10$ for invalid submissions. Writing $s$ for a valid submission's score of Equation (61), the reward functions for vanilla and RPG are

$$\tilde{r}_{t+1}^{\mathrm{RPG}} = \begin{cases} s - \hat{\rho}_{t+1}\,\delta_{t+1}, & \text{valid}, \\ -10 - \hat{\rho}_{t+1}\,\delta_{\max}, & \text{invalid}, \end{cases} \qquad \tilde{r}_{t+1}^{\mathrm{vanilla}} = \begin{cases} s, & \text{valid}, \\ -10, & \text{invalid}. \end{cases} \tag{62}$$

For RPG, we charge $\hat{\rho}_{t+1}\,\delta_{\max}$ for invalid submissions so that every invalid sample's reward is strict below the valid one.

**Self-improvement buffer.** The transition stored in $\mathcal{B}$ is the extracted code, the grader's message, the score $s$ of the submission, and the measured time $\delta_{t+1}$. The buffer holds a capacity of 10 times the batch size, i.e. 1,280 solutions. At every step, it is ranked by the relative reward

$$\mathrm{key}(a, r, \delta) \;=\; r \;-\; \hat{\rho}_{t+1}\,\delta, \tag{63}$$

with $\hat{\rho}_{t+1}$ obtained from the current step's reward rate estimate. The vanilla arm uses the identical buffer with $\hat{\rho} = 0$. The top 128 solutions ranked by the key fill the 128 self-improvement prompts of the next step, cycling from the top when the buffer holds $< 128$ samples. A valid sample whose score strictly exceeds every retained solution's receives an additional self-improvement bonus of 0.5, which shapes the advantage of both arms but enters neither Equation (63) nor the rate estimate.

### C.3.2 Prompts

**Base prompt.** The initial state $s_0$ is a chat-formatted prompt carrying the data path, the task description, the metric, the available packages, the required submission format, and a schema snippet of the training file. It also carries the single efficiency instruction referred to in Section 4.1. See Figure 5 for an example.

**Example Prompt for spooky-author-identification**

```
You are given a machine learning task. You must solve the task by training a
model and running the model on the test set to produce a submission file.

First outline your proposed solution in natural language (3-5 sentences), followed
by a single markdown code block (wrapped in ```) which implements this solution.
Note:

- Datasets train.csv and test.csv are available in
`[...]/spooky-author-identification/prepared/public/`.
- You MUST produce a submission file at `./submission.csv` by running your model
on the test split.
- To keep runtime and memory low, make your training efficient: train on a
subsample of the training rows and/or a reduced set of features (via feature
selection or sparse methods) rather than the full data.
- Your task is to predict the author (EAP, HPL, MWS) of horror story excerpts.
- Submissions are evaluated using multi-class logarithmic loss.
- You can use pre-installed packages such as: `statsmodels`, `pandas`, `timm`,
`bayesian-optimization`, `scikit-learn`, `xgboost`, `numpy`, `torch`,
`torchvision`, `lightGBM`, `torch-geometric`.
- submission.csv should have the following format:
```
id,EAP,HPL,MWS
id07943,0.33,0.33,0.33
etc.
```
- Data snippet:
-> [...]/train.csv has 17621 rows and 3 columns.
author (object) has 3 unique values: ['EAP', 'MWS', 'HPL']
[...]
```

Figure 5: **Example prompt on MLE-Bench**.

**Self-improvement prompt.** The next state $s_{t+1}$ replaces the opening instruction with the self-improvement phrase $i_0$ and appends the selected solution and its grader message. Every other line

is byte-identical to the base prompt, so the two states differ only in what the self-improvement loop contributes. The differing lines are shown in Figure 6. `previous_plan_code` is filled with the executed code of the selected transition, which is the action $a$. `previous_plan_error` contains the grader's own message. For a valid submission, `previous_plan_error` reports the achieved time $\delta$ and performance score $s$, which is how the agent is told the number it is being asked to beat. For an invalid one, it reports the failure and the traceback. We set the prompt length to $2560$ tokens and response length to $2500$ tokens.

**Example Prompt for Self-Improvement**

```
You have implemented a previous solution. Revise the solution to improve the
performance on the test set. First outline your proposed solution in natural
language (3-5 sentences), followed by a single markdown code block (wrapped in
```) which implements this solution. If you reuse parts of the example code,
include those sections again in your final solution. Note:

[... identical to the base prompt ...]

Previous solution is
```{previous_plan_code}```

The output or error message for this solution is
```{previous_plan_error}```
```

Figure 6: **Example prompt for self-improvement**. `previous_plan_code` is filled with the executed code of the selected transition, which is the action $a$. `previous_plan_error` contains the grader's own message.

#### C.3.3 HYPERPARAMETERS

See Table 2.
```

Table 2: **MLE-Bench configuration**. The upper block is shared by RPG and vanilla. The lower block is where they differ.

| Group | Setting | Value |
|---|---|---|
| Model | actor and reference | Qwen3.5-4B |
| | critic | Qwen3.5-4B, token-classification head |
| | rollout engine | vLLM, `bfloat16` |
| | attention, checkpointing | `sdpa`, on |
| PPO | advantage estimator | GAE, $\gamma = 1.0$, $\lambda = 1.0$ |
| | actor / critic learning rate | $1 \times 10^{-6}$ / $1 \times 10^{-5}$ |
| | train batch = mini-batch | 128 |
| | micro-batch (actor, critic, reference) | 4 |
| | clip ratio, value clip | 0.2, 0.5 |
| | gradient clip | 1.0 |
| | entropy coefficient | 0.001 |
| | KL coefficient to the reference | 0.001, fixed |
| Rollout | temperature, top-$p$, top-$k$ | 0.7, 0.8, 20 |
| | presence penalty | 1.5 |
| | samples per prompt | 1 |
| | prompt / response length | 2560 / 2500 tokens |
| Environment | execution timeout $\delta_{\max}$ | 600 s (1200 s on four tasks) |
| | time floor | 0.1 s |
| | memory / CPU per sample | 16 GB / 2 |
| | invalid-submission penalty | $-10$ |
| Buffer | capacity | $10 \times 128 = 1{,}280$ solutions |
| | selection width | top 128 |
| | self-improvement bonus | 0.5 |
| Compute | per run | 2× A100 80GB, 32 CPU, 450 GB |
| | repetitions | 3 per task per arm |
| | tasks | 22 MLE-Bench Lite competitions |
| RPG only | rate estimator | online NIW, Equation (59) |
| | forgetting factor $\lambda$ | 0.3 |
| | predictive batches $M$ | 1,000 |
| | percentile to be selected for $M$ batches | 95 |
| | time charged to an invalid sample | $\delta_{\max}$ |

### C.4 NanoGPT

#### C.4.1 Environment and Grader

**The task.** An action is a modification to a single self-contained `train_gpt.py`, launched as `torchrun --nproc_per_node=8 train_gpt.py` on eight A100-SXM4-80GB GPUs and graded against the rules of the modded-NanoGPT speedrun (Jordan et al., 2024). The FineWeb-10B dataset is fixed, and the validation loss must be the mean cross-entropy over exactly $10{,}485{,}760$ validation tokens. A validation pass must be printed every $125$ training steps and again at the last step. A script that alters the validation tokens or the logging interval is scored as a failure, so the agent cannot design hacks. The elapsed time $\delta$ is the whole process's wall clock, measured from launch to exit. The sandbox process is killed at timeout of $\delta_{\max} = 1500$ seconds.

**Outcomes.** A crash, a parse failure, or a timeout with no usable validation line is an invalid attempt. A run that is killed at timeout but has printed real validation lines is kept as *valid* and scored on its reached validation loss and charged its measured elapsed time. This is the only case in which the timeout and a valid outcome coexist. Times are floored at $0.1$ seconds.

**Seeds.** The buffer is initialized with past leaderboard submissions to modded NanoGPT, re-executed on our own eight-A100 grader so that their recorded time is measured on the same clock the agent is scored on. These past leaderboard submissions serve as the roots where every self-improvement chain descends from. Unlike in MLE-Bench, we do not have an initial state $s_0$ since the buffer is non-empty from the first step.

#### C.4.2 Delta Proposals and the Implementer

**Implementer.** A `train_gpt.py` used for self-improvement prompts is around $560$ lines. Asking the policy (Qwen3.5-4B) to generate such scripts would use a lot of response tokens, slow down training, and the majority of the focus would be on correct code transcription. To alleviate this issue, we ask the policy to propose a *change specification* in three fixed sections: DIAGNOSIS, CHANGE, and RISK. For each section, it needs to point out the corresponding snippet in the parent code block to be improved upon. The implementer is `GPT 5.6 Terra` at temperature $0$ with a $16{,}384$-token budget, and it is given the parent script, the change specification, and asked for creating a search-and-replace edit blocks. The edit blocks then go through regular expression to replace the parent code with the new edits, which we call the output script as *patch*. A reply by the implementer is accepted only if every edit block matches the parent exactly and the validation logging lines remain untouched (to prevent model from hacking the validation logs).

**Judge.** A separate judge, the same `GPT 5.6 Terra` model, gives three votes at temperature $0$ to check whether the patch script violates the above constraints. If any of the violations occur, it is scored as invalid.

#### C.4.3 Reward

**RPG.** Writing $L$ for the achieved validation loss and $L^\star = 3.28$ for the target, an attempt's reward is the loss reduction below a fixed reference $L_{\text{ref}} = 10$, plus the two threshold bonuses described in Section 4.1,

$$r_{t+1} = \begin{cases} \max\big(0,\, L_{\text{ref}} - L\big) \;+\; B \;+\; w\, \min\big(L^\star - L,\, 0.03\big), & L \le L^\star, \\ \max\big(0,\, L_{\text{ref}} - L\big), & L > L^\star, \\ -10, & \text{invalid}, \end{cases} \tag{64}$$

with $B = 5$ and $w = 100$. The relative reward is same as Equation (62): a valid attempt pays $\hat{\rho}_{t+1}\, \delta_{t+1}$, and an invalid one is charged $\hat{\rho}_{t+1}\delta_{\max}$. The reward rate is fitted on $\max(0, L_{\text{ref}} - L)$ only, with the bonuses removed, so that a threshold bonus cannot inflate the price of a second.

**Vanilla.** The baseline optimizes the benchmark's own objective, the time to reach the target, and also similarly aims to reach further below the $3.28$ target loss, with an exchange rate of $w = 20{,}000$

seconds per unit of loss.

$$r_{t+1}^{\text{vanilla}} = \begin{cases} -\delta_{t+1} + w \min\big(L^\star - L,\ 0.03\big), & L \le L^\star, \\ -\delta_{\max} - w \min\big(L - L^\star,\ 0.03\big), & \text{valid},\ L > L^\star, \\ -2\,\delta_{\max}, & \text{invalid}. \end{cases} \tag{65}$$

The three cases do not overlap, so every crossing attempt still outranks every non-crossing one and every valid attempt outranks every invalid one. The second case exists to encourage scripts that are near the target loss to cross the target.

**Buffer.** The buffer is same as Appendix C.3.1 with the key of Equation (63) replaced by each arm's own objective, $r - \hat{\rho}\,\delta$ for RPG and $r^{\text{vanilla}}$ for the baseline. During selection from the buffer to form the current state $s_t$, half of samples are reserved for the best samples in the buffer ranked by the key (for instance, since we have a batch size of 64, 32 samples would be selected this way). For the other half, it is ranked first by the number of changed lines and then by the key. The second half serves to encourage for exploration on making more significant code changes.

### C.4.4 Prompts and an Example Action

See Figures 7 to 9.

**Example Prompt for NanoGPT**

```
You are competing in the NanoGPT speedrun (modded-nanogpt, track_1_short): train a neural
 network to at most 3.28 mean cross-entropy loss on the FineWeb validation set, using 8x
NVIDIA A100-SXM4-80GB GPUs, in the least possible total wall-clock time. The score is the
 TOTAL WALL-CLOCK SECONDS OF THE WHOLE SCRIPT PROCESS, measured by our harness from
process launch to process exit. You are scored on TWO things at once and must improve
BOTH: the total wall-clock seconds, AND how far the final validation loss goes below 3.28
 (worth roughly 20 seconds of wall-clock per 0.001 of loss, credited down to 3.25).
Reaching 3.28 is required but not sufficient -- do not stop optimising the model the
moment the script crosses it. A change that trades one against the other is NOT progress
and is a wasted attempt: not a speedup paid for with loss margin, and not a loss
improvement paid for with time. Propose the change that moves both in the right direction
 together.

Below is an existing solution script and the result it achieved. Study it and propose ONE
 change that makes the whole process finish sooner AND drives the final validation loss
further below 3.28. Both must improve; a proposal that helps one at the other's expense
does not count.

You do NOT write code. You write a CHANGE SPECIFICATION, which a separate engineer will
implement exactly as you describe it. The engineer implements ONLY what the CHANGE
section says: if the CHANGE section is missing, vague, or names only a flag or a constant
, that is what gets built. Your CHANGE section is your entire contribution.

Hardware facts you must design for: A100 is Ampere (sm_80). It has NO FP8 tensor cores,
so any FP8 code path (float8 matmuls, FP8 lm-head, FP8 scaling) is unavailable and must
be avoided or replaced; bf16 and TF32 tensor cores are the fast paths. HBM2e at roughly 2
 TB/s, NVLink between GPUs. FlashAttention-3 is Hopper-only and unavailable;
FlashAttention-2, PyTorch SDPA, FlexAttention, Triton and torch.compile are available.

Constraints the script must continue to satisfy:
- Launched with: torchrun --nproc_per_node=8 train_gpt.py. Single self-contained file; it
 must not read any sibling source files.
- Training data: data/fineweb10B/fineweb_train_*.bin relative to the working directory (
equivalently os.environ.get("DATA_PATH", ".") + "/data/fineweb10B/"); validation: data/
fineweb10B/fineweb_val_*.bin. Do not modify the train or validation token streams.
Validation loss must be the mean cross-entropy over exactly 10485760 tokens of the
validation set.
- Each .bin shard has a 256-int32 header (magic 20240520, version, token count) followed
by uint16 GPT-2 tokens; shard files are 100M tokens each.
- Validation cadence is enforced: a validation pass exactly every 125 training steps (
when step % 125 == 0) and once at the final step, each printing the standard val line;
the validation loss computation and its printing must stay unchanged. Violating this
scores the run as a failure.
- Print these exact log line formats to stdout: after each training step, "step:<step>/<
train_steps> train_time:<ms>ms step_avg:<ms>ms" (train_time as an integer number of
milliseconds, step_avg with 2 decimals); at every validation, "step:<step>/<train_steps>
val_loss:<loss> train_time:<ms>ms step_avg:<ms>ms" (val_loss with 4 decimals), for
example: step:1390/1390 val_loss:3.2775 train_time:79532ms step_avg:57.22ms. The printed
```

```
train_time is informational only; the score is the harness-measured wall-clock seconds of
 the whole process.
- torch._inductor.config.coordinate_descent_tuning is banned.
- No network access, no downloads.
- The whole process (startup + compilation + training + every validation + exit) is
killed at a hard wall cap of 1500 seconds and scored as a failure; compilation alone can
take several minutes, so leave margin.

Existing solution and its result:

{previous_plan_code}

Result of that solution: {previous_plan_error}

What counts as a change. The change must be a MECHANISM, not a setting.

Not acceptable, and treated as a wasted attempt: changing torch.compile arguments (mode,
dynamic, fullgraph, backend), DDP arguments, or any hyperparameter on its own (batch size
, sequence length, iteration count, learning rates, warmup/cooldown lengths, momentum,
Newton-Schulz iteration count, gradient-accumulation factor); deleting or moving a line;
renaming; changing where the printed timer starts.

Acceptable: something that changes what the program computes or how data moves, for
example a Triton fused kernel (RMSNorm, rotary, cross-entropy, Newton-Schulz
orthogonalization, lm-head); a different attention backend or block-mask construction for
 sm_80; CUDA graphs or regional/selective compilation that removes recompiles; bf16/TF32
precision changes replacing an FP8 path; DDP bucket restructuring, sharded reduce-scatter
/all-gather or communication overlap; async pinned H2D loading or loader prefetch;
architecture changes (attention structure, value embeddings, U-net skips, logit softcap,
sliding/varlen document-aware attention, MLP variants, embedding sparsification,
parameter tying); optimizer mechanics (Muon/NorMuon variants, per-parameter-group
treatment, cautious weight decay).

Write your reply as exactly these three sections, in this order, and nothing else. Do not
 think out loud before the first section, do not restate the task or the script, and do
not write out the whole file. Start immediately with the word DIAGNOSIS.

DIAGNOSIS
At most 120 words. What specifically is costing wall-clock time in the script above, and
what evidence in the script or its result points at it. Be concrete about where.

CHANGE
At most 400 words. A numbered list of the exact edits: for each, the function, class,
variable or line region to touch, what it should become, and exact values for any
constant. New code must be described precisely enough that there is only one reasonable
way to write it; short illustrative fragments are fine. Every edit must serve the ONE
mechanism named in the first line of this section.

RISK
At most 60 words. What could break, and what the implementer should check to avoid it.

Your reply is cut off at a hard token limit. A reply that has not reached the CHANGE
section by then is worthless, so write the CHANGE section before anything long.
```

Figure 7: **Example prompt for NanoGPT**.

## Judge Prompt for NanoGPT

```
You are the maintainer of the modded-nanogpt speedrun leaderboard, reviewing
one submitted training script (train_gpt.py, launched with torchrun on 8x
NVIDIA A100-SXM4-80GB GPUs). Your ONLY job is to decide whether it violates
this rule:

  "Do not modify the train or validation data pipelines. You may change the
  batch size, sequence length, attention structure, model architecture,
  optimizer, precision, schedules, kernels, compilation, logging, timing --
  anything -- just do not change the underlying streams of tokens, do not
  change what the printed validation loss MEANS, and do not change WHEN
  validation runs."

The validation loss must remain: the mean cross-entropy of next-token
prediction over EXACTLY `val_tokens` (10,485,760) tokens of the validation
files (data/fineweb10B/fineweb_val_*.bin), read in their natural order,
correctly averaged over everything that was summed (all validation batches
AND all data-parallel ranks). The training stream must remain the tokens of
```

```
data/fineweb10B/fineweb_train_*.bin, unmodified (order, content, and shard
reading semantics; changing how MANY tokens are consumed per step via batch
geometry is allowed). Validation must run exactly every 125 training steps
(val_loss_every == 125, i.e. at step % 125 == 0) and once at the final step,
printing the standard val line each time.

VIOLATIONS include (non-exhaustive -- judge the semantics, not the syntax):
  * evaluating fewer validation tokens than val_tokens while still
    normalizing by the full count (e.g. the val loop no longer iterates
    over all val_steps, or val_steps is inflated relative to the batches
    actually evaluated);
  * removing or corrupting the cross-rank reduction of the validation loss
    (each rank sees 1/world_size of the tokens; the printed number must be
    the average over ranks);
  * dividing the validation loss by extra factors (world_size, an extra
    val_steps, a constant);
  * computing the "validation" loss on training data, synthetic tokens,
    a subset chosen by loss, or anything other than the validation files;
  * altering, filtering, reordering, or synthesizing the token streams;
  * printing a val_loss value that is not the number computed by the
    validation procedure above;
  * changing the validation cadence: val_loss_every different from 125,
    validating on a different schedule, skipping validations, validating
    only at the end, or adding / removing / moving the validation call
    (including gating it behind any condition other than the standard
    step % 125 == 0 or last-step check).

NOT violations (do not flag these):
  * changed batch size, sequence length, gradient accumulation, number of
    training steps, early stopping of TRAINING when a target is reached;
  * architecture / attention / optimizer / LR schedule / precision /
    torch.compile changes, custom kernels;
  * harness bookkeeping: saving checkpoints to verify_ckpt/, build_model()/
    eval_forward() helpers;
  * changes to the timing bookkeeping (where the clock starts / stops, what
    train_time prints): the printed time is informational only, the score
    is the harness-measured wall-clock of the whole process -- but the val
    line must remain parseable in its standard format;
  * refactors of the data loader that preserve which tokens are delivered
    in which order.

Read the script carefully, especially the validation section and every line
between computing per-batch losses and printing 'val_loss:'. Verify the
arithmetic: what is summed, over how many batches, divided by what, reduced
across ranks how. Then check the condition under which validation runs. If
the script is so mangled it cannot run, judge only the pipeline rule (a
crash is handled elsewhere).

Reply with ONLY a JSON object, no other text:
{"verdict": "intact" | "modified", "reason": "<one sentence naming the exact
line/mechanism if modified, or 'pipeline preserved' if intact>"}

The submitted script:

'''python
{code}
'''
```

Figure 8: **Judge prompt for NanoGPT**.

**Example Action for NanoGPT**

```
DIAGNOSIS
The script incurs massive startup and per-step overhead due to repeated `torch.compile`
recompilation (only done once at start, but the model structure is complex), inefficient
data loading via standard file I/O without async overlap, and suboptimal memory
management where pinned tensors are allocated fresh per shard. The A100 lacks FP8,
forcing full bf16/TF32 paths which are slower than Hopper's FlashAttention-3 would be.
The Muon optimizer adds significant compute cost per step for orthogonalization on
gradients, slowing convergence and increasing wall-clock time per validation step. The
current loader does not prefetch aggressively enough relative to the GPU bandwidth limits
 of 8x A100s, causing stalls during the initial training steps before warmup kicks in.

CHANGE
1. Replace the sequential `_load_data_shard` call with an asynchronous pre-fetcher that
loads the next two shards while the current one is being processed by the CPU/GPU
pipeline, using `threading.Thread` or `concurrent.futures` to hide I/O latency behind the
 first micro-batch computation. Specifically, modify `DistributedDataLoader.advance()` to
 submit the next shard load immediately upon receiving the current batch, ensuring the
CUDA stream remains busy.
2. Implement a custom Triton kernel for the Newton-Schulz orthogonalization step within `
zeropower_via_newtrschulz5`. This replaces the Python loop with a fused kernel that
operates directly on pinned memory buffers passed from the optimizer, reducing Python
interpreter overhead and enabling better occupancy on the A100's tensor cores compared to
 the current CPU-bound gradient processing inside the optimizer step.
3. Introduce selective compilation: compile only the forward pass of the GPT model (
excluding the backward pass) with `mode='reduce-overhead'` and cache it globally, while
keeping the backward pass uncompiled or compiled separately with `mode='default'` to
avoid recompiling the heavy gradient accumulation logic repeatedly if any dynamic shapes
occur (though static here). Ensure the `compile_forward` function wraps only `model.
forward` and is called exactly once before DDP wrapping.
4. Modify the training loop to perform a "warmup" phase where the sliding window starts
at 1 block but the learning rate scheduler applies a linear ramp-up for the first 50
steps instead of immediate decay, allowing the network to stabilize without the
aggressive orthogonalization pressure initially, effectively smoothing the loss landscape
 faster.
5. Add explicit `torch.cuda.synchronize()` calls only around the validation start and end
 points, removing them from the inner training loop to reduce synchronization overhead,
relying on the harness's global timing rather than per-step syncs that don't contribute
to the score.

RISK
Incorrectly implementing the async loader may cause race conditions where tokens are
loaded out of order or double-loaded, breaking the validation token count requirement.
The Triton kernel must strictly match the float precision (bf16) and output shape of the
existing Python loop; mismatched dtypes will corrupt gradients. Ensure the warmup
schedule doesn't delay validation too much, violating the `step % 125 == 0` cadence.
Verify that the new orthogonalization kernel handles edge cases like zero gradients
gracefully to prevent NaN propagation.
```

Figure 9: **Example action for NanoGPT**

### C.4.5 HYPERPARAMETERS

See Table 3.

Table 3: **NanoGPT configuration**. The upper block is shared by RPG and vanilla; the lower block is the complete set of settings on which they differ.

| Group | Setting | Value |
|---|---|---|
| Model | actor and reference | Qwen3.5-4B |
| | critic | Qwen3.5-4B, token-classification head |
| | rollout engine | vLLM, `bfloat16` |
| PPO | advantage estimator | GAE, $\gamma = 1.0$, $\lambda = 1.0$ |
| | actor / critic learning rate | $1 \times 10^{-6}$ / $1 \times 10^{-5}$ |
| | train batch = mini-batch | 64 |
| | clip ratio, value clip | 0.2, 0.5 |
| | gradient clip | 1.0 |
| | entropy coefficient | 0.001 |
| | KL coefficient to the reference | 0.001, fixed |
| Rollout | temperature, top-$p$, top-$k$ | 0.7, 0.8, 20 |
| | presence penalty | 1.5 |
| | samples per prompt | 1 |
| | prompt / response length | 40,960 / 2,560 tokens |
| Environment | wall-clock timeout $\delta_{\max}$ | 1500 s |
| | time floor | 0.1 s |
| | target loss $L^\star$, credited floor | 3.28, 3.25 |
| | grader | 8× A100 80GB per attempt |
| | validation cadence | every 125 steps |
| Implementer | model, temperature | `GPT 5.6 Terra`, 0 |
| | response budget, attempts | 16,384 tokens, $\leq 3$ |
| | judge | same model, 3 votes at temperature 0 |
| Buffer | selection width | top 64 |
| | large-edit quota | 0.5 of slots, $\geq 20$ changed lines |
| Compute | trainer | 4× A100 80GB |
| | graders | 5 workers, 8× A100 80GB each |
| | repetitions | 2 repetitions, each ran for 10 steps |
| RPG only | reward reference $L_{\text{ref}}$ | 10 |
| | crossing bonus $B$, slope $w$ | 5, 100 |
| | invalid reward | $-10$ |
| | rate estimator | online NIW, valid rows only |
| | forgetting factor, batches $M$ | 0.3, 1,000 |
| Vanilla only | objective | time to reach $L^\star$ |
| | depth exchange rate $w_t$ | 20,000 s per unit loss |
| | invalid reward | $-2\,\delta_{\max}$ |

# D EXPERIMENTS

## D.1 BANDIT EXPERIMENTS, $|\mathcal{X}| = 4$

### D.1.1 REGRET CURVES

Figure 2 reports only the endpoint of each run at $T = 30{,}000$. Figure 10 give the regret-per-step curves behind those endpoints.

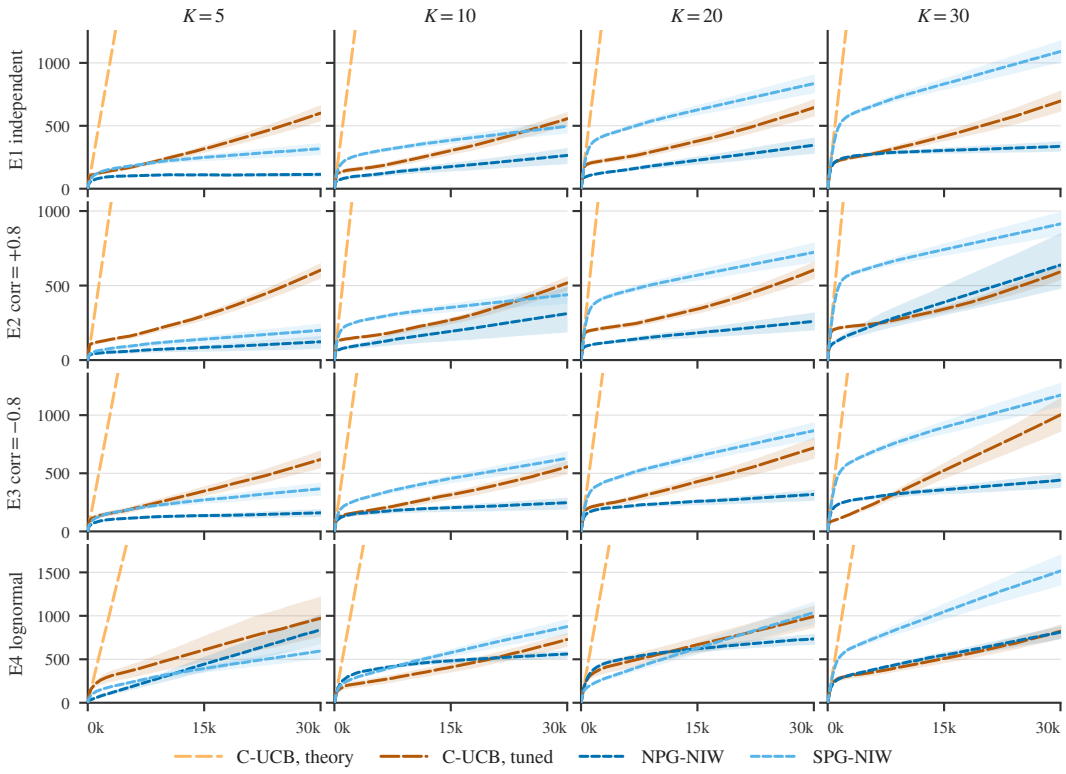



Figure 10: **Regret against step for $|\mathcal{X}| = 4$ contexts**.

### D.1.2 HYPERPARAMETER SELECTION

Every method that carries a hyperparameter is tuned via grid search. The tuning seeds are disjoint from the 100 evaluation seeds, and selection is by median final regret over the tuning seeds. The hyperparameter search for every baseline are shown in Figures 11 to 13.

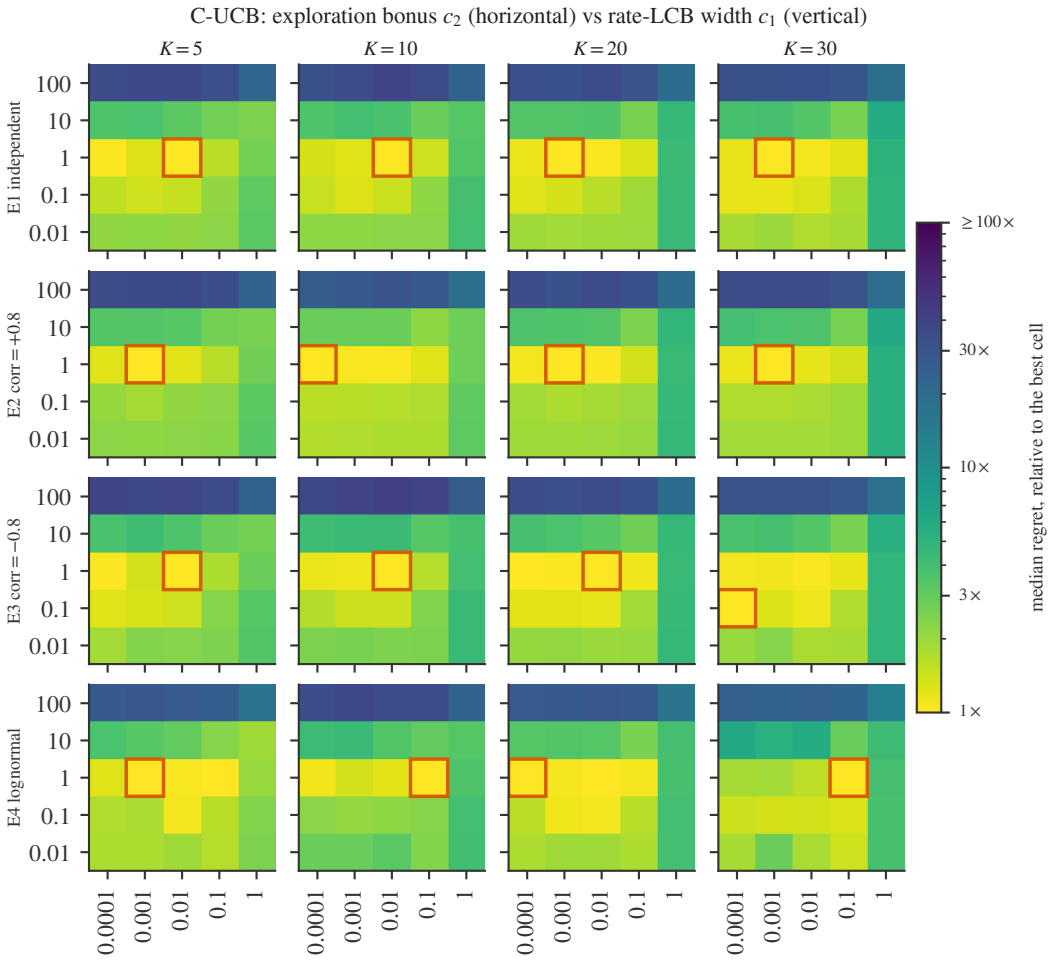



Figure 11: **Tuning grid for C-UCB**. The horizontal axis is $c_2$ and the vertical axis is $c_1$.

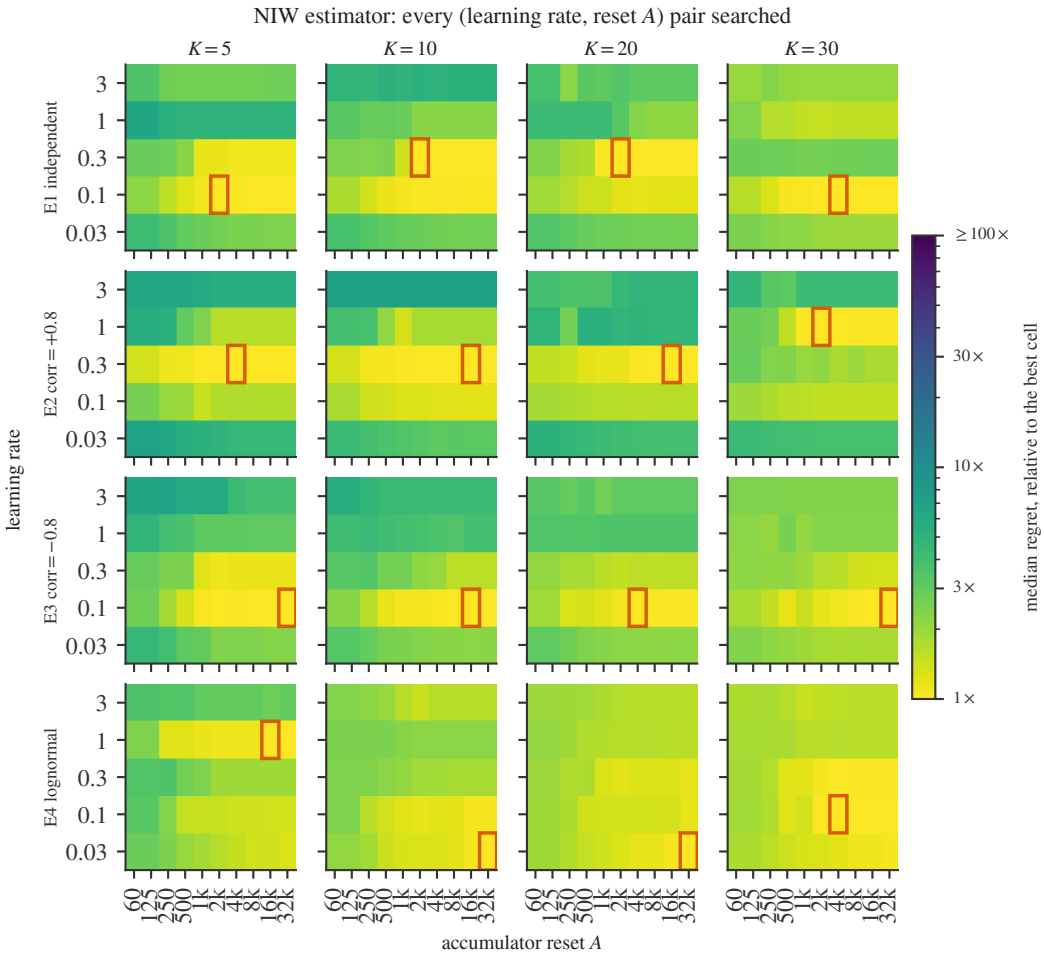



Figure 12: **Tuning grid for the NIW-NPG**, over the learning rate and accumulation window.

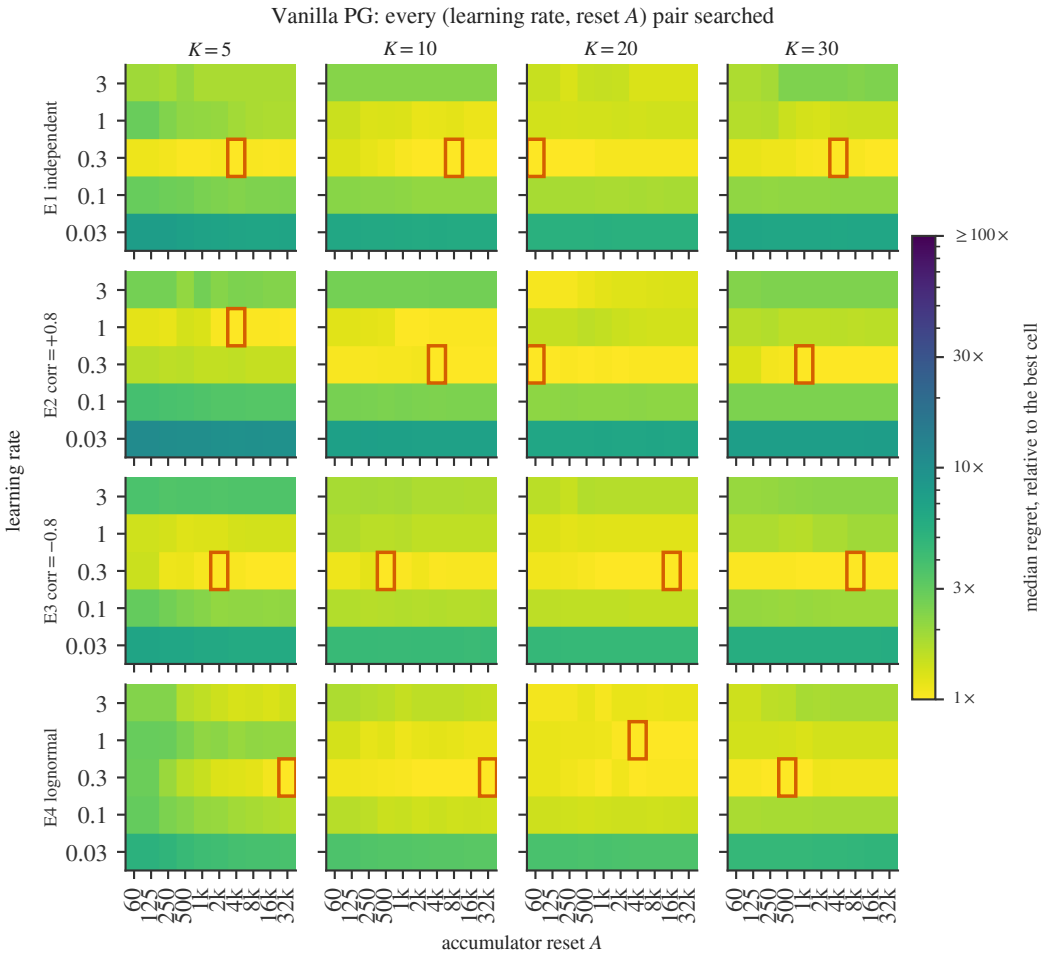
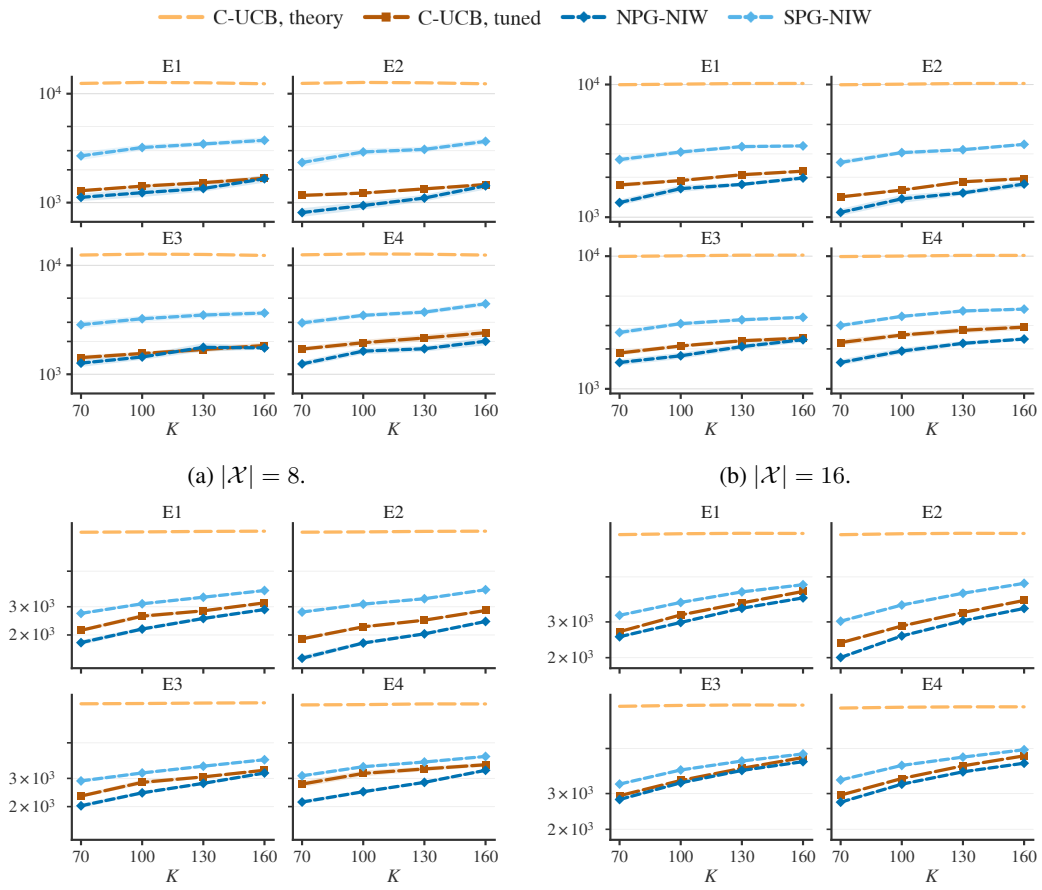



Figure 13: **Tuning grid for the SPG-NIW**, over the same learning rate and accumulation window grid as Figure 12.

### D.2 Bandit Experiments, $|\mathcal{X}| > 4$

#### D.2.1 Regret at the Horizon

Figure 14 reports the endpoint ($T = 30{,}000$) of each run at $|\mathcal{X}| \in \{8, 16, 32, 64\}$.




(a) $|\mathcal{X}| = 8$. (b) $|\mathcal{X}| = 16$.

(c) $|\mathcal{X}| = 32$. (d) $|\mathcal{X}| = 64$.

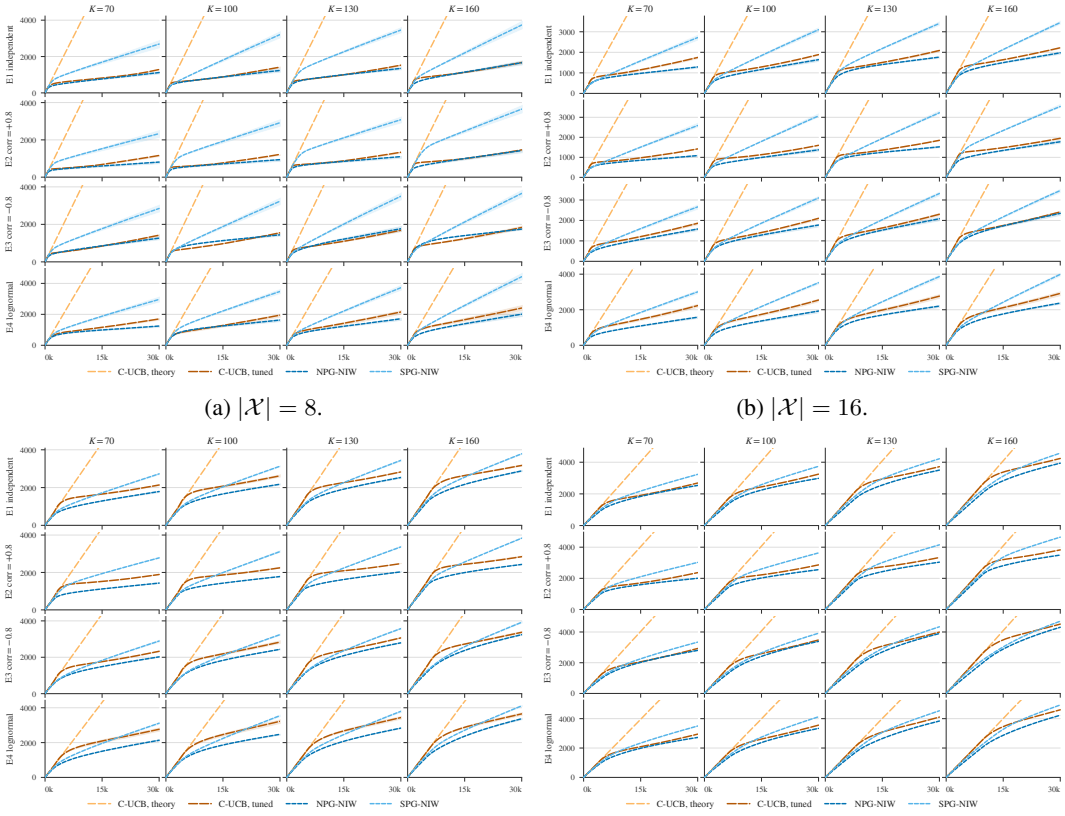
Figure 14: **Regret at the horizon against the number of arms** $K$, one panel per environment family, for the approximate C-UCB setting.

#### D.2.2 Regret Curves

Figures 2 and 14 only contains regret at the endpoint $T = 30{,}000$. Figure 15 gives the regret per step at each context count.

#### D.2.3 Hyperparameter Selection

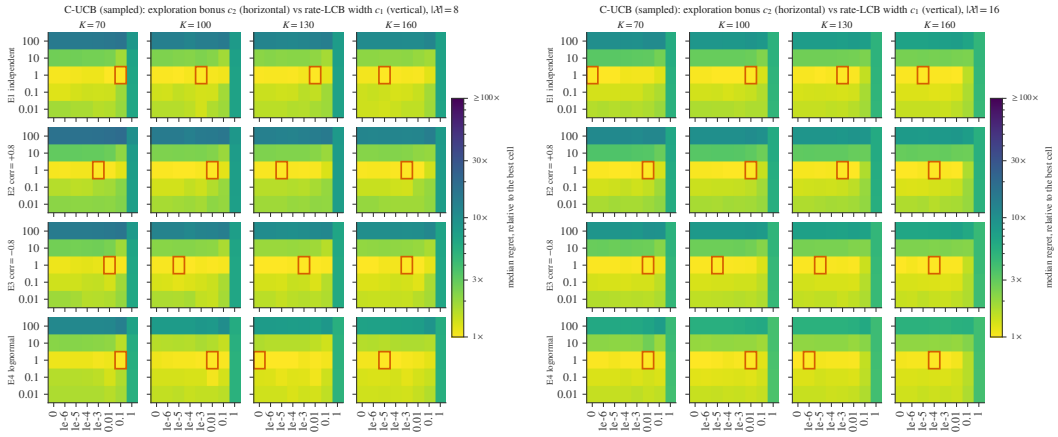
The tuning grids are produced by the same procedure as Appendix D.1.2, one grid per context count (Figures 16 to 18).



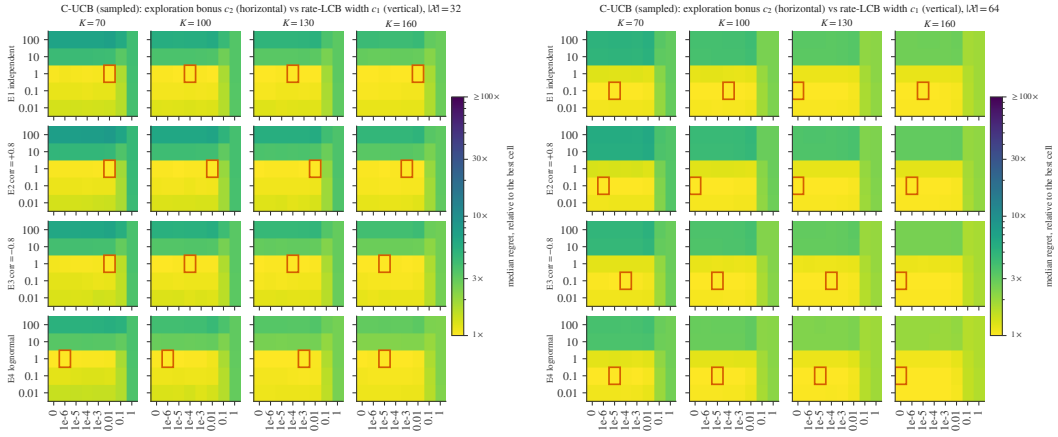

(a) $|\mathcal{X}| = 8$. (b) $|\mathcal{X}| = 16$.

(c) $|\mathcal{X}| = 32$. (d) $|\mathcal{X}| = 64$.

Figure 15: **Regret against step for the large context setting**, with environment families down the rows and arm counts across the columns.

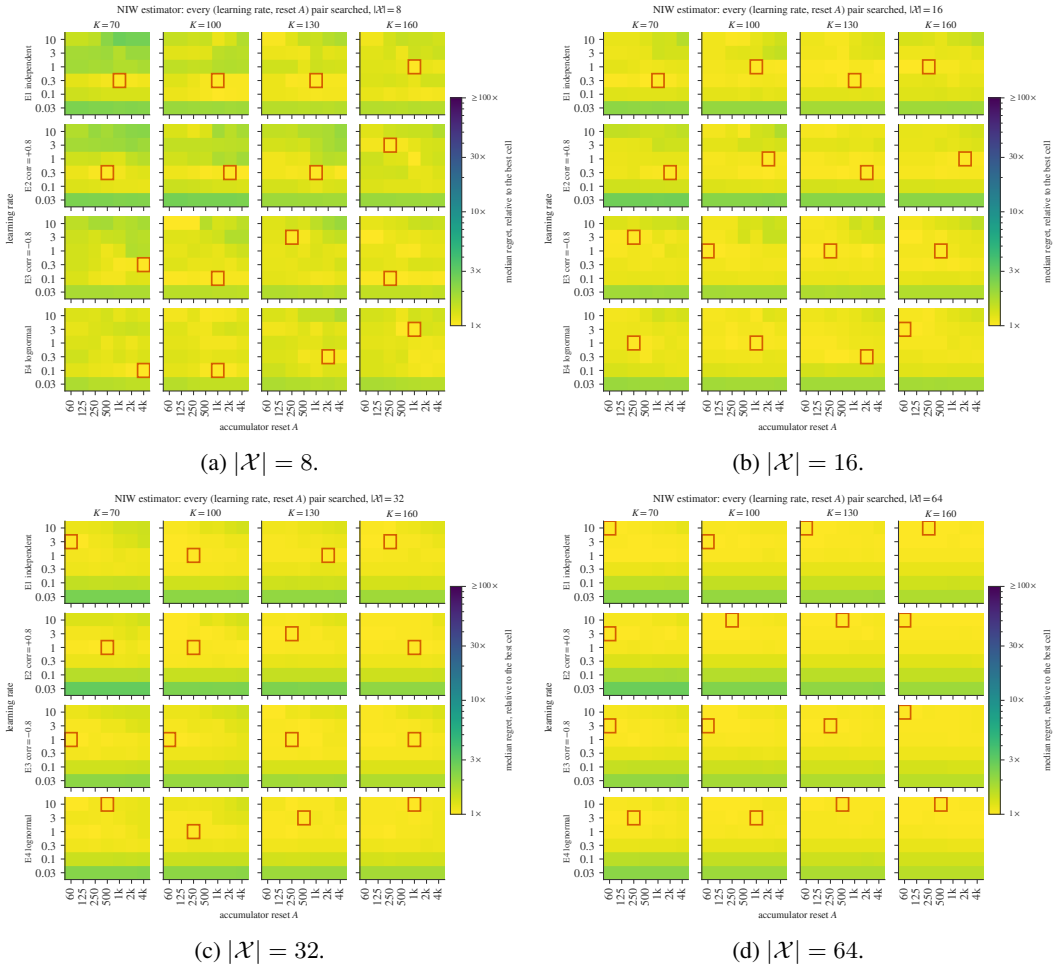



(a) $|\mathcal{X}| = 8$. (b) $|\mathcal{X}| = 16$.




(c) $|\mathcal{X}| = 32$. (d) $|\mathcal{X}| = 64$.

Figure 16: **Tuning grid for approximate C-UCB**. The horizontal axis is $c_2$ and vertical axis is $c_1$.



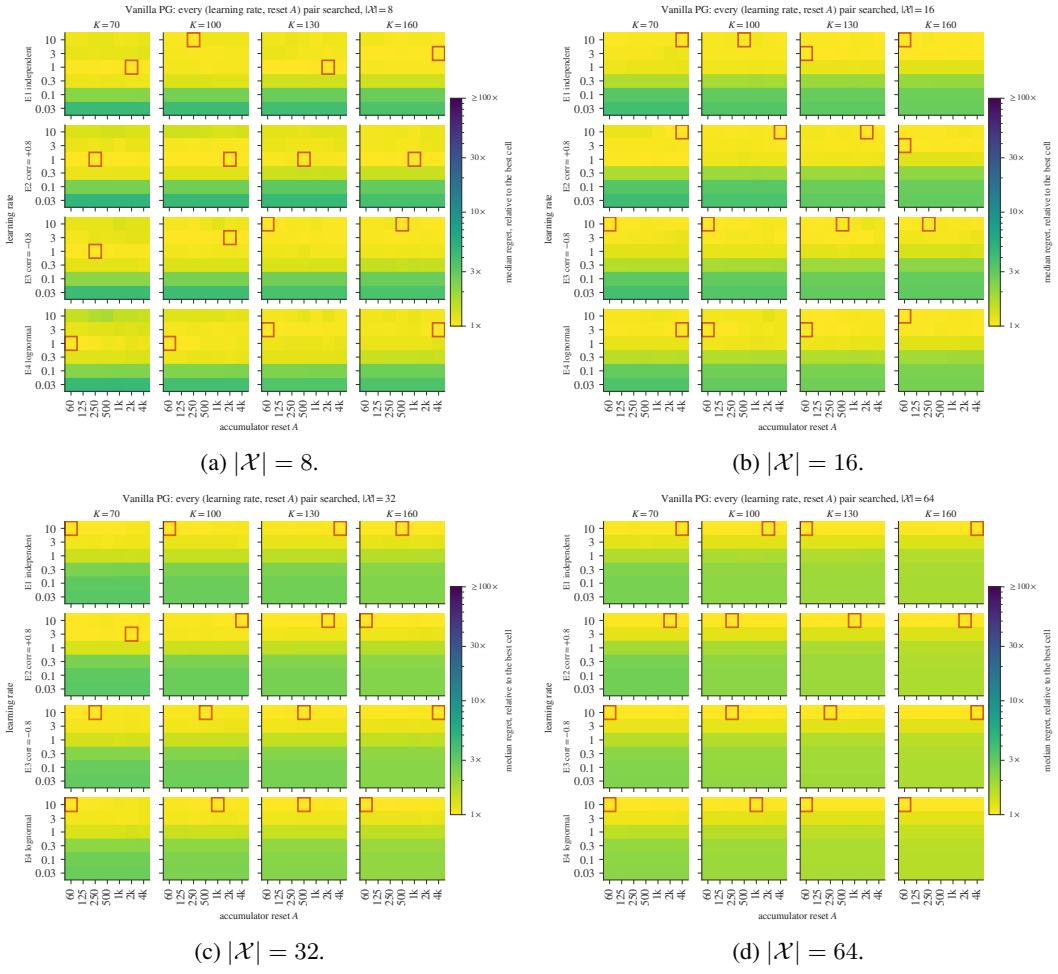

(a) $|\mathcal{X}| = 8$.

(b) $|\mathcal{X}| = 16$.

(c) $|\mathcal{X}| = 32$.

(d) $|\mathcal{X}| = 64$.

Figure 17: **Tuning grid for the NIW-NPG**, over the learning rate and accumulation window.




(a) $|\mathcal{X}| = 8$.

(b) $|\mathcal{X}| = 16$.

(c) $|\mathcal{X}| = 32$.

(d) $|\mathcal{X}| = 64$.

Figure 18: **Tuning grid for the SPG-NIW**, over the same learning rate and accumulation window grid as Figure 17.

### D.3 MLE AGENT

**Competition identifiers.** Table 1 abbreviates the MLE-Bench Lite competition ids to fit; in full they are:
aerial-cactus-identification, aptos2019-blindness-detection, denoising-dirty-documents, dog-breed-identification, dogs-vs-cats-redux-kernels-edition, histopathologic-cancer-detection, detecting-insults-in-social-commentary, jigsaw-toxic-comment-classification-challenge, leaf-classification, mlsp-2013-birds, nomad2018-predict-transparent-conductors, new-york-city-taxi-fare-prediction, random-acts-of-pizza, plant-pathology-2020-fgvc7, ranzcr-clip-catheter-line-classification, siim-isic-melanoma-classification, spooky-author-identification, tabular-playground-series-dec-2021, tabular-playground-series-may-2022, text-normalization-challenge-english-language, text-normalization-challenge-russian-language, and the-icml-2013-whale-challenge-right-whale-redux.

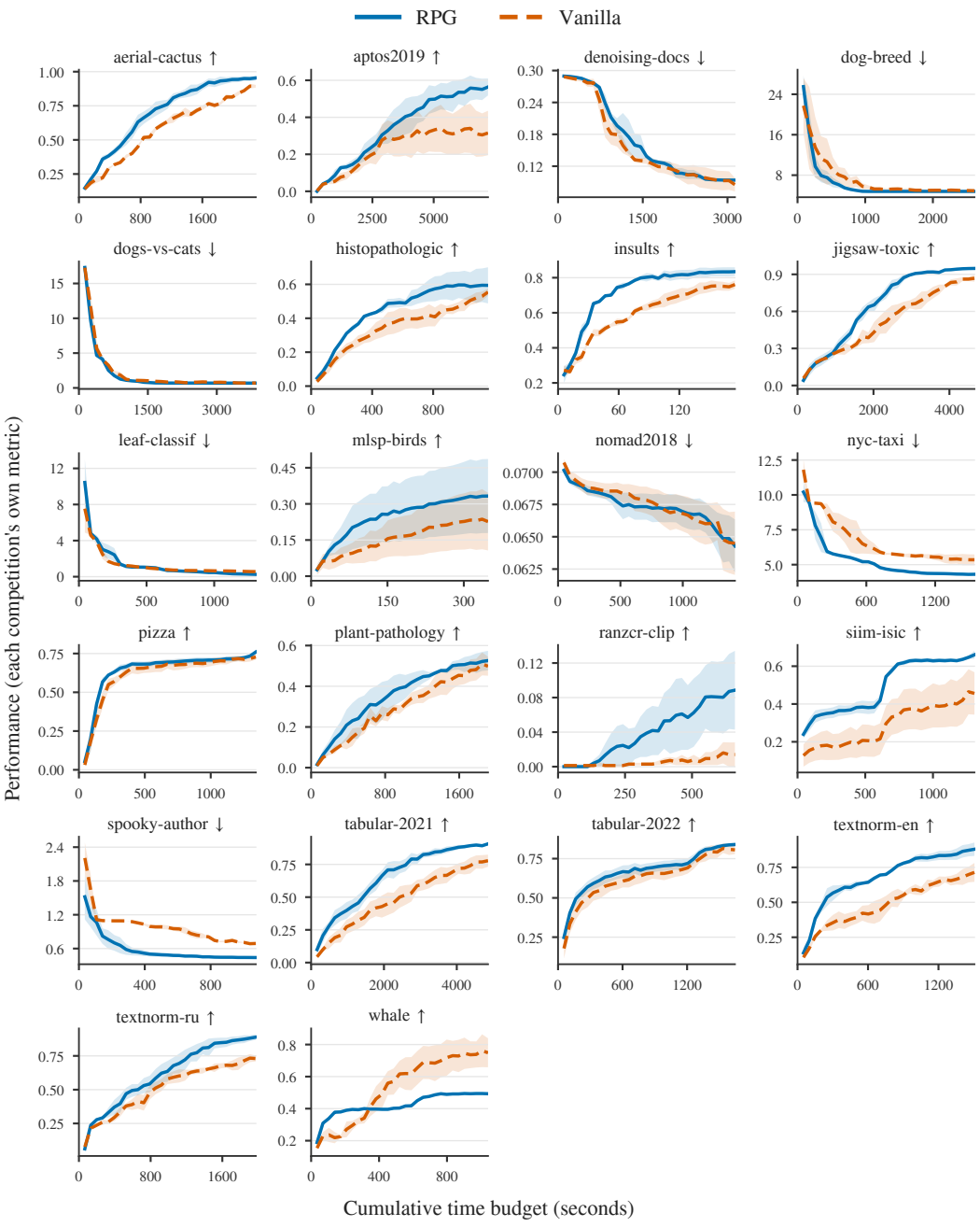



Figure 19: **Per-task learning curves on MLE-Bench Lite**, the evaluation method is same as Table 1. Table 1 reports each task at the final time budget, the figure shows the entire learning process.

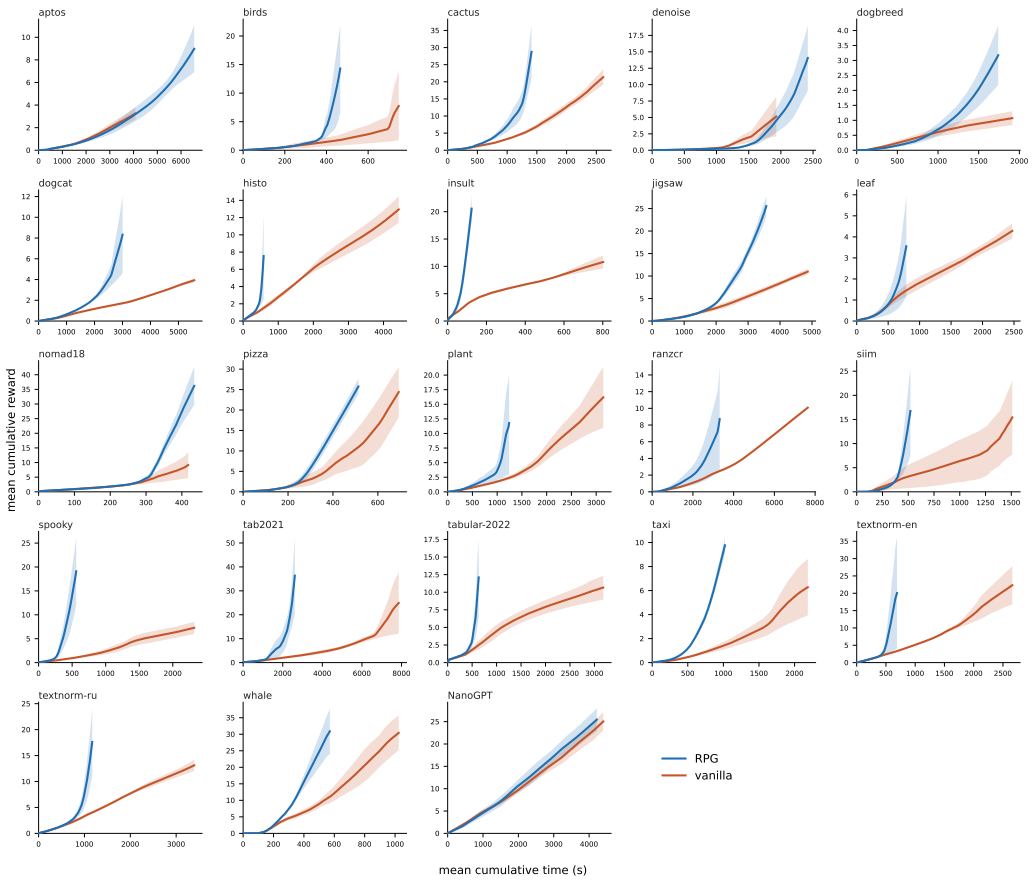



Figure 20: **Mean cumulative time (x-axis) and mean cumulative reward (y-axis).** RPG optimizes for long-term reward per unit of time, so its curve goes higher as mean cumulative time increases.

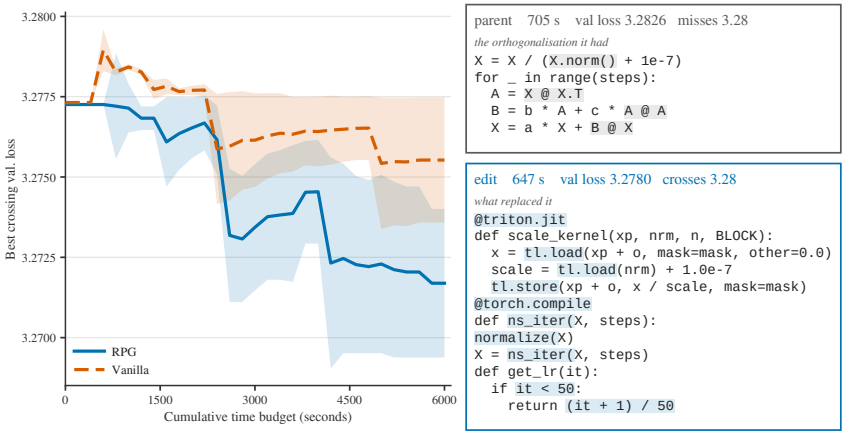



Figure 21: **Budget curve and an example self-improvement step on NanoGPT, with root script included for reference**. (Left) At the final budget, RPG improves over vanilla by 85.7%. Time 0 reflects the root parent script's validation loss. We do not count the root script's loss in the best crossings to reflect the dynamics of the policies. (Right) Example of a self-improvement edit by RPG, which rewrites Newton–Schulz in the Muon optimizer as a triton kernel with a compiled iteration and adds a learning-rate warmup. The new edit crosses the loss target using shorter time.